RICE UNIVERSITY

Foundations for Automatic, Adaptable Compilation

by

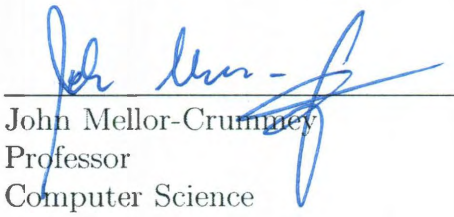
Jeffrey Andrew Sandoval

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

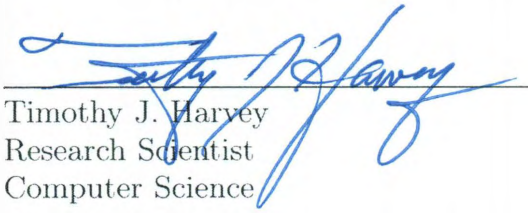
APPROVED, THESIS COMMITTEE:

Keith D. Cooper, Chair
L. John and Ann H. Doerr Professor
Computational Engineering



John Mellor-Crummey
Professor
Computer Science

Joseph R. Cavallaro
Professor
Electrical and Computer Engineering



Timothy J. Harvey
Research Scientist
Computer Science

HOUSTON, TEXAS
MARCH 2011

ABSTRACT

Foundations for Automatic, Adaptable Compilation

by

Jeffrey Andrew Sandoval

Computational science demands extreme performance because the running time of an application often determines the size of the experiment that a scientist can reasonably compute. Unfortunately, traditional compiler technology is ill-equipped to harness the full potential of today's computing platforms, forcing scientists to spend time manually tuning their application's performance. Although improving compiler technology should alleviate this problem, two challenges obstruct this goal: hardware platforms are rapidly changing and application software is difficult to statically model and predict. To address these problems, this thesis presents two techniques that aim to improve a compiler's adaptability: automatic resource characterization and selective, dynamic optimization.

Resource characterization empirically measures a system's performance-critical characteristics, which can be provided to a parameterized compiler that specializes programs accordingly. Measuring these characteristics is important, because a system's physical characteristics do not always match its observed characteristics. Consequently, resource characterization provides an empirical performance model of a system's actual behavior, which is better suited for guiding compiler optimizations than a purely theoretical model. This thesis presents techniques for determining a system's data cache and TLB capacity, line size, and associativity, as well as instruction-cache capacity.

Even with a perfect architectural-model, compilers will still often generate sub-optimal code because of the difficulty in statically analyzing and predicting a pro-

gram's behavior. This thesis presents two techniques that enable selective, dynamic-optimization for cases in which static compilation fails to deliver adequate performance. First, intermediate-representation (IR) annotation generates a fully-optimized native binary tagged with a higher-level compiler representation of itself. The native binary benefits from static optimization and code generation, but the IR annotation allows targeted and aggressive dynamic-optimization. Second, adaptive code-selection allows a program to empirically tune its performance throughout execution by automatically identifying and favoring the best performing variant of a routine. This technique can be used for dynamically choosing between different static-compilation strategies; or, it can be used with IR annotation for performing dynamic, feedback-directed optimization.

Acknowledgments

I would first like to thank Keith Cooper, my thesis advisor, for his guidance and insight throughout my time at Rice. I have really enjoyed working with him and learning from him. My thesis committee members—John Mellor-Crummey, Joe Cavallaro, and Tim Harvey—have also shared their guidance and expertise. I thank them for their comments and suggestions, which have helped improve my work considerably.

I would also like to acknowledge the contributions and support of many other Rice students, faculty, and alumni, including: Penny Anderson, Rajkishore Barik, Thomas Barr, Yuri Dotsenko, Mary Fletcher, John Garvin, Yi Guo, Anshu Das Gupta, Mack Joyner, Cheryl McCosh, Dave Peixotto, Apan Qasem, Raghavan Raman, Steve Reeves, Arnold Schwaighofer, Kamal Sharma, Ray Simar, Nathan Tallent, Dr. Richard Tapia, Reid Tatge, Linda Torczon, Todd Waterman, Anna Youssefi, Ryan Yang Zhang, and Yuan Zhao.

I especially appreciate the friendship and generosity of Chirag Patel and Luke Probst—they made my frequent trips to Houston comfortable and enjoyable.

Finally, I am indebted to my family for all of their love and inspiration. My wife, Monica, has been very patient with my intermittent progress, empathetic with my frequent frustrations, and wonderful support throughout my entire project. My parents, Gerald and Joyce, and my siblings, Elizabeth, Steven, and Patrick, have always taught me to do my best and to not give up. I am thankful for their confidence in me, and I appreciate all of the ways in which they have shaped my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer modeling and simulation is an integral component of modern research in the natural sciences, allowing scientists to conduct virtual experiments that are too costly or time-consuming—or even too dangerous—to perform in the real world [3]. Nuclear fission, earthquakes, large-scale warfare exercises, automobile crashes, and wind tunnel experiments to reduce drag on automobiles—without the expense of building physical models or prototypes—are all examples for which computer simulation has become an indispensable tool for scientific discovery. These applications are particularly demanding with regards to performance, because an application's running time often determines the level of detail or the size of experiment that the scientist can reasonably compute. That is, the progress of the scientific research is often limited by the available computing power. To further exacerbate this problem, traditional compiler technology is ill-equipped to harness the full potential of modern high-performance computing platforms, forcing scientists to spend time manually tuning their application performance rather than advancing their scientific research. Because this kind of tuning is inherently platform specific—and because it is usually applied, by hand, to the application's source code—the tuning must be re-applied for each new architecture—and, in some cases, each new architecture version—on which the application will run. Consequently, this process is tedious, time-consuming, and

error-prone; it is precisely the kind of work that is well-suited to computer automation in the form of compiler technology. Unfortunately, existing compiler technology faces two challenging obstacles in this context. First, hardware platforms are rapidly changing with advances in computer architecture, and achieving full performance requires careful targeting of each system's unique performance-related characteristics. Second, the behavior of application software is difficult to statically model and predict, which hinders the compiler's ability to generate efficient machine code; this problem will only grow worse as modern scientific software becomes larger and more complex. This thesis presents two foundational approaches for addressing these problems: automatic resource characterization and selective, dynamic optimization. These complementary techniques aim to improve the adaptability of compilers, both from the perspective of application software and the underlying hardware.

1.1 Automatic Resource Characterization

Automatic resource characterization alleviates one of the challenges caused by rapid changes in hardware. Historically, compilers are manually ported to new architectures as they become available. Although some efforts aim to automate this process through formal architecture specifications and automatic code generators, the success of these efforts is in quickly producing a working and correct compiler rather than producing a powerful, optimizing compiler. Correctness is unquestionably more important than performance. But, achieving performance still requires manual tuning for each new architecture and model. In addition, modern microprocessor designs are advancing at such a rapid rate that a microprocessor can become obsolete before a mature compiler is even available. Ideally, a compiler should automatically tune itself for a new architecture, replacing slow and expensive human effort with relatively fast and inexpensive computer automation. Supporting this approach requires an automatic way to produce accurate values for performance-related characteristics

of an underlying system. Although values for some of these characteristics are publicly documented, this thesis argues that relying on such documentation is insufficient for several reasons. First, each manufacturer follows a different documentation standard, requiring manual examination of a processor's documentation. In addition, not all characteristics are clearly or completely disclosed—some characteristics, such as the branch predictor's behavior or the cache replacement policy are closely guarded trade secrets. Second, characteristics can vary widely across the myriad architecture variants, even differing between models of the same processor family; manually maintaining complete and accurate records of this information becomes unwieldy. Third, and most importantly, a system's physical characteristics often do not always match its observed characteristics. Consequently, this thesis argues that empirically measuring characteristics for each system is the best solution. Automatic resource characterization provides an empirical performance model of a system's actual behavior, rather than its theoretical behavior. From a compiler's perspective, this observed model is best suited for optimization decisions.

This thesis presents a suite of micro-benchmarks that empirically determine a subset of a system's performance-related characteristics. The micro-benchmarks are designed to be *portable*, because they must run on a wide variety of current and future systems. They are *focused*, so as to clearly distinguish between various characteristics. They are fully *automatic* and able to determine each characteristic without human intervention. Finally, they are designed to be *robust*, so we can be confident that their results are reasonable. This thesis focuses on automatically characterizing a system's memory hierarchy, because efficient use of the memory hierarchy has a large impact on performance. Chapter 2 presents techniques for characterizing a system's data cache and TLB capacity, line size, and associativity. Chapter 3 presents similar techniques for characterizing a system's instruction-cache capacity and determining which levels of cache contain both data and instructions.

1.2 Selective, Dynamic Optimization

Even with perfect information about the underlying system, compilers will still often generate sub-optimal machine code for an application because of the difficulty in statically analyzing and predicting a program's behavior. Therefore, this thesis proposes *dynamic optimization* as a reinforcing technique for cases in which static compiler technology fails to deliver adequate performance.

Traditionally, compilers perform program optimization entirely ahead-of-time with respect to program execution. We refer to these compilers as *static compilers*, because they analyze and transform the program's source code without knowledge of how the program behaves at runtime. Although this approach amortizes the cost of a single compilation with the benefit of repeated execution, static compilers are unable to exploit runtime context and behavior during optimization. As scientific software grows larger and more complex, the community is beginning to embrace various software engineering techniques to improve productivity [7]. Separate compilation, generic interfaces, and component-based frameworks all promise to improve productivity. Unfortunately, these constructs also impede static compiler optimization, effectively reducing the static compiler's ability to produce efficient executables. This trend, along with the knowledge that not all performance-critical data is available at compile time, makes a strong case for dynamic compilation.

Dynamic compilers apply optimization while a program is running, allowing specialization to be driven by runtime context and behavior. A static compiler must assume that all program control paths are possible and equally likely, while a dynamic compiler can observe which paths are most likely and which paths never (or rarely) occur. In this manner, a dynamic compiler can adapt its optimization decisions for the particular context in which a program is running. Further, a dynamic compiler can continually monitor and re-optimize a program during its entire duration of execution. This effectively allows a compiler to tune and re-tune an application for

any changes in behavior that may arise during execution.

The main challenges with dynamic compilation are performance overhead, portability, and outcome unpredictability. Overhead is a problem because any invocation of a compiler at runtime increases the program's overall processor time; thus, a compiler should only be invoked when it expects the benefits of optimization to outweigh the costs. Past attempts at dynamic optimization have been faced with high overheads that must be overcome before any benefit is realized [11]. Portability is a problem because most systems either target a single programming language—such as Smalltalk [19], Self [13], or Java [5, 12, 39, 47]—or a single computer architecture—such as Dynamo [8], Dynamo RIO [11], ADORE [34], or Pin [35]. Outcome unpredictability refers to the problem that dynamic-compilation systems do not often evaluate the impact of their actions. For interpreted languages, it is generally safe to assume that compiled code will perform better than interpreted code. For statically compiled code, however, it is often less clear that a particular dynamic optimization will guarantee program speedup. In practice, sometimes an optimization improves performance and sometimes it degrades performance [16].

This thesis introduces a new technique, called *intermediate-representation (IR) annotation*, that enables low-overhead, cross-language, *selective* dynamic compilation. IR annotation is implemented as a static compilation transformation that generates a fully-optimized native binary tagged with a higher-level compiler intermediate representation of itself, similar to the technique described by Tatge *et al.* [49]. IR annotation reduces the overhead of dynamic compilation because the optimized binary achieves the speed of native execution immediately upon invocation; however, the IR annotation allows aggressive runtime optimization that can selectively target performance bottlenecks. IR annotation supports cross-language runtime optimization by leveraging a compiler's common IR format. Finally, IR annotation allows the dynamic compiler to de-couple itself from the underlying architecture—all op-

timizations are performed with a portable IR before being transformed to native code. Thus, most platform-specific implementation details are pushed into a single component: the code generator.[1] These design features of IR annotation will allow it to support a low-overhead dynamic compilation framework that increases a compiler's ability to adapt to the dynamic context and behavior of a running application. Chapter 4 presents the details of IR annotation, as well as an experimental case-study supporting the feasibility of this technique.

To improve optimization accountability for dynamic compilers, this thesis introduces another new technique called *adaptive code-selection*, which allows a running program to automatically identify and favor the best performing variant of a routine. This technique has several novel uses. It can be paired with IR annotation and a dynamic compiler to allow quick and objective evaluation of the impact of runtime optimization decisions, offering an opportunity to dynamically reverse or alter decisions that resulted in performance degradation. Or, it can simply enable a running application to dynamically choose from between several pre-compiled versions of a routine, allowing dynamic selection of the best static-compilation strategy. Chapter 5 presents the details of adaptive code-selection and an experimental case-study that illustrates the benefits of this technique.

1.3 Overview

Resource characterization and dynamic optimization are complementary technologies that build a solid foundation for adaptable compilation. The two technologies attack problems that occur at different scales. Resource characterization improves a static compiler's ability to quickly adapt to the rapid changes in underlying hardware, which generally occur at a rate of every 12-18 months. Dynamic optimization improves a runtime compiler's ability to quickly adapt to the unpredictable changes in an appli-

[1] Of necessity, the code generator must know those details to perform it's primary function.

cation's context and behavior, which can occur at a rate of every few minutes to every few hours or days for long running programs. These two technologies are cooperative. Resource characterization improves a compiler's ability to model a computing platform and to predict a program's performance, in light of various optimization decisions. Dynamic optimization, on the other hand, improves a compiler's ability to re-tune a running application based on empirical observations of a program's behavior and performance characteristics. Static compilation, with an improved architectural model, is able to generate more efficient code that is less in need of dynamic tuning. This allows the dynamic compiler to focus more exclusively on regions of an application that are inherently difficult to model and predict statically. However, a more accurate performance model should also improve a dynamic compiler's ability to quickly tune an application, since achieving good performance should require fewer iterations in the dynamic tuning-feedback loop.

The remainder of this document presents the details of automatic resource characterization, IR annotation, and adaptive code-selection. The dissertation is divided into four main components: characterizing the data cache, covered in Chapter 2; characterizing the instruction cache, covered in Chapter 3; enabling dynamic compilation with IR annotation, covered in Chapter 4; and, adaptively selecting the best code-variant at runtime, covered in Chapter 5. Each of these chapters presents the motivation and implementation details specific to that topic, as well as experimental results to support the approach. Each chapter ends with an overall summary of the work and a discussion about future research that arises from this dissertation. Lastly, Chapter 6 concludes the dissertation with a brief review of each topic and an assessment of how it fits into the overall theme of this dissertation.

Chapter 2

Data Cache Characterization

Application performance on today's multicore processors is often limited by the performance of the system's memory hierarchy. To achieve good performance, the code must be carefully tailored to the detailed memory structure of the target processor. That detailed structure varies widely across different models of the same ISA. Thus, performance is often limited by the compiler's ability to tailor the program's behavior in model-specific ways.

Current practice in compiler construction makes model-dependent optimization difficult. Simply put, the compiler has no way of determining the memory hierarchy parameters of the target system. Static information provided by the vendor or the operating system can serve as a starting point, but the compiler needs a more nuanced view of the hierarchy. An Intel T9600 processor, with a 6 MB L2 cache that is shared between the two cores and between the instruction and data cache hierarchies, is unlikely to perform well if both processors run code that was blocked for the full 6 MB of cache.

A better approach is to derive, at compiler installation time, system dependent parameters for the memory hierarchy. This chapter presents a set of techniques that discover, empirically, the effective capacities and other parameters of the various levels

in the memory hierarchy, both cache and TLB. By *effective capacity*, we mean the amount of memory at each level that one processor can use before the average latency begins to rise.

The distinction between the hardware capacity and the effective capacity is critical. On many current processors, the higher levels of cache exhibit latency curves that begin to rise considerably before the actual hardware cache boundary. Several factors contribute to this behavior: sharing between the instruction and data cache hierarchy, sharing among cores on multicore systems, and other uses for upper level cache, in particular the operating system's page table. In addition, the use of physically-mapped caches at the upper levels means that portable techniques cannot be assured that a contiguous array maps into contiguous memory. All of these effects contribute to reduced effective cache-capacities in L2 and higher caches.

A compiler that performs blocking of memory accesses to improve behavior should achieve better results using these effective cache sizes than it would using the manufacturer's published numbers. The published numbers show hardware capacity, but not the portion of that capacity available to an executing program. The compiler should optimize the application to that latter number, the effective capacity.

The techniques described in this chapter examine the memory hierarchy from the perspective of data accesses. Since the main loop for all variants of these data cache benchmarks is very small—containing fewer than 50 instructions on a typical architecture—the impact of instruction accesses can safely be ignored. Thus, the results in this chapter reflect the data memory hierarchy; Chapter 3 presents techniques for characterizing the instruction memory hierarchy by performing similar experiments from the perspective of instruction accesses. Chapter 3 also shows how to detect *unified* levels in the memory hierarchy, which contain both data and instructions.

This chapter builds on a long line of prior work, described in Section 2.1. It extends

that work in several important ways. Our focus is on robust micro-benchmarks and automated analysis to interpret the results. To provide clean and reliable data, our tools use novel reference strings that isolate cache and TLB behavior from one another. They employ a disciplined approach to time measurement that reduces timer noise. Section 2.2 describes the micro-benchmarks and their implementations. To analyze the data, the tools use a sophisticated multi-step technique that provides a consistent interpretation of micro-benchmark results. Section 2.3 presents the details of our analysis technique. Finally, our goal is to build a portable, robust toolset. Section 2.4 describes our experience using the tools to characterize more than twenty distinct processors and processor models.

2.1 Related Work

Memory hierarchy characterization has a long history in the literature. The same basic idea underlies most of the related work: carefully construct parameterized data footprints that stress particular characteristics of the memory hierarchy, and measure the performance of accessing that footprint across a range of parameters. The observable performance differences should correspond to thresholds in the underlying memory hierarchy. The distinction between various approaches lies in the following categories:

Coverage Which cache and TLB characteristics does the tool measure: capacity, latency, line size (or page size) and associativity? Does the tool handle both virtually and physically mapped caches?

Approach What data footprints does the tool use for each characteristic? Does it use a single access pattern for multiple characteristics or does it separate and isolate individual effects?

Portability How portable is the tool? Does it rely on any languages, tools, or libraries that are not widely available? Does it use other platform-specific information?

Analysis Does the tool require any manual intervention, human interpretation, or does it automatically interpret the benchmark results? How robust is the analysis in the face of noise or unexpected data?

Durability Is the tool prepared to cope with future advances in compilers and architectures? What changes might cause (or have already caused) the tool to function improperly or produce an incorrect result.

Much of the memory characterization work derives from Saavedra and Smith [43, 44]. They probe the cache and TLB hierarchy with a Fortran benchmark that measures the performance of accessing an array of length N with a stride of s. They generate plots with varied values of N and s and manually interpret the results to determine the cache and TLB capacity, line size (page size), and associativity. They use a single benchmark to determine all characteristics, which requires careful disambiguation between various effects. In contrast, our work uses separate access patterns to clearly distinguish between various cache and TLB effects; we provide a robust analysis technique that handles results from both virtually and physically mapped caches. Saavedra and Smith did not specifically address physically mapped caches, although they recognized that the results would be less clear because the virtual-to-physical mapping is not guaranteed to be contiguous.

McVoy and Staelin present `lmbench` [37], a suite of micro-benchmarks that measure various system characteristics, including the memory hierarchy. Their basic approach resembles the work of Saavedra and Smith; but, instead of traversing data footprints through strided array accesses, they use a linked list to minimize the impact of compiler optimizations. Unfortunately, today's hardware prefetchers would easily

detect the constant memory stride embedded within the linked list. Also, `lmbench` did not measure TLB capacity because the caches were small enough to be fully addressed without any TLB misses. In contrast, modern processors often employ both multi-level caches and multi-level TLBs, where the largest cache footprints exceed the coverage of the largest TLB.

This criticism is not limited to `lmbench`; much of the early work in this area was effective on contemporary systems, but subsequent hardware advances have necessitated changes in approach. Today's challenges, such as sophisticated hardware prefetchers, multi-level caches and TLBs, shared caches, multi-core processors, and non-uniform cache access (NUCA) designs, did not exist 10 or 15 years ago. To their credit, McVoy and Staelin made the astute observation that their approach was designed for contemporary (*i.e.,* 1993 - 1995) systems; they suggested that future architectural advances could present challenges to their approach. It would certainly be naive for us to claim that our work is immune to future changes in architecture or technology. However, our intent is to build tools that will work on systems for the near term future and to design techniques that have lasting value. Thus, we have made a focused effort to minimize the impact that the compiler and hardware can have on the effectiveness of our benchmark.

In 2005 Yotov *et al.* introduced X-RAY [51, 52], a significant advance in the state-of-the-art of resource characterization. They address both algorithmic and implementation deficiencies in prior work, and they provide a strong theoretical foundation for their methods. X-RAY characterizes caches by simultaneously detecting capacity and line size, essentially probing the cache to determine its *shape.* Their approach characterizes caches efficiently and accurately. X-Ray requires contiguous mapping in the cache, which limits it to virtually-mapped caches or systems that support page coloring. To measure physically mapped caches, X-Ray can use superpages, a feature that remains, today, a portability problem. Our tool characterizes physically mapped

caches. Also, X-RAY determines actual cache sizes, where our tools measure effective cache sizes.

González-Domínguez *et al.* presented Servet [27], a portable micro-benchmark suite for characterizing cache capacity, shared cache and NUMA topology, memory access bottlenecks, and distributed memory communication costs. Servet detects virtually mapped caches (*i.e.*, L1 cache) with a gradient search. For physically mapped caches the gradient proved problematic, so they estimate the actual hardware cache sizes with a probabilistic binomial model suggested by Fraguela *et al.* [24]. Our approach differs from Servet in that we employ a more powerful global analysis to determine the *effective* cache size rather than the actual cache size for all levels of the cache and TLB. Our experimental results indicate that the effective cache size is often much smaller than the actual cache size. Servet counteracts hardware prefetching by using a large (1 KB) stride, following Saavedra and Smith [43]. We use randomization to defeat prefetchers, an approach that we expect to remain viable for the foreseeable future. Finally, we characterize a system's TLB hierarchy while Servet does not.

Both Servet and P-RAY [23], an extension to X-RAY, characterize sharing and communication aspects of multi-core clusters that we do not address in this work. Our techniques for improving cache characterization methodology from the perspective of a single core is orthogonal to the work on characterizing shared resources.

Dongarra *et al* [22] present a characterization tool that leverages hardware performance counters to measure actual cache and TLB misses instead of observing elapsed time alone. This approach certainly allows easier disambiguation between cache and TLB effects, but it is limited to systems that provide the appropriate performance counters. We considered this approach but discarded it because of the lack of standardized support for performance counters.

Our work differs from previous work in several ways. We can characterize cache capacity, latency and line size for both physically and virtually mapped caches; we only

characterize associativity for virtually mapped caches. We measure TLB capacity. We do not measure page size, since it is provided through the portable POSIX `sysconf` interface. Our tool requires a C compiler and a POSIX environment. Our tools do not currently measure TLB associativity, but our framework should easily support the addition of an access pattern to measure it. Our global analysis of benchmark results is more robust than previous approaches, which either relied on human intervention or performed localized automatic analysis. Our approach should be less sensitive to selecting appropriate thresholds. Finally, our characterization benchmark finds *effective* cache parameters, while previous work strives to determine actual hardware parameters. Experimental results on all of our tests systems indicate that the L1 effective capacity equals the hardware capacity. However, for higher levels in a system's memory hierarchy our results indicate that the entire hardware capacity is not usually available for an application's use—cache sharing, physical mapping, and system page-tables all reduce the cache's effective capacity. Our characterization results correctly reflect this behavior by reporting effective cache capacities that are smaller than the theoretical hardware values.

2.2 Portable Micro-benchmarks

Our tools conduct a variety of tests on the memory hierarchy. This section describes tests that measure the capacity and latency for the various levels of cache and TLB, along with associativity for the L1 cache, and line sizes for all levels of cache. The tools are designed to be portable and robust. They rely on a standard C compiler and the POSIX operating system interfaces. We build on POSIX interfaces for key components, such as an accurate timer and an allocator that returns page-aligned arrays.[1]

[1]All of the tests in this chapter use page-aligned arrays to eliminate one source of variation between runs.

This section begins with a simple test, which we call the *gap test*, that measures characteristics of the L1 data cache. We measure L1 using a specialized test for several reasons. First, the L1 measurements are the easiest to make. On all the machines we have tested, the L1 cache is core-private and virtually mapped. These features allow a simple test to obtain precise measurements of the hardware characteristics. We explain the gap test first because it exposes most of the complications that arise in measuring memory hierarchy effects.

After the gap test, this section presents two tests that measure properties of the higher levels of the memory hierarchy: one that measures capacity of the higher level caches and another that measures TLB capacities. Next, it discusses the techniques that we use to find line sizes and associativities. The final subsection describes a cache-oblivious access order that will work without knowledge of the L1 data cache parameters.

2.2.1 Gap Test

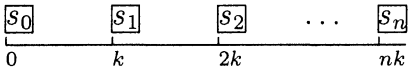
The first step that our tools take is to measure the characteristics of the L1 data cache. Later tests use the L1 line size.[2] An accurate measurement of L1 line size lets us eliminate the effects of spatial locality within an L1 line; that, in turn, makes the measurements taken in those later tests more clear.

To analyze L1 cache characteristics, we use a particularly simple test. It accesses a set of n locations spaced k bytes apart. We call this set a *reference string*. We describe the reference strings for the gap test with a tuple $G(n,k,o)$ where n is the number of locations, k is the number of bytes between the start of those locations, and o is an offset added to the start of the last location in the set. The reference string $G(n,k,0)$ generates the following locations.

[2]If we cannot determine L1 line size, the later tests assume a line size of `sizeof(int)`, which may reduce the clarity of the results. The cache-oblivious ordering, described in Section 2.2.6 can also work without L1 line size.

```
baseline ← time for the G(2,LB,0) reference string
for n ← 1 to MaxAssociativity
  for k ← LB to UB
    t ← time for the G(n,k,0) reference string
    if (t > baseline)
      L1Size ← n × k
      L1Assoc ← n - 1
      break out of both loops
for offset ← 1 to page size
  t ← time for the G(n,k,offset) reference string
  if t = baseline
    L1LineSize = offset / 2
    break out of loop
```

Figure 2.1: Pseudocode for the Gap Test



The reference string *G(n,k,4)* would move the n^{th} location out another four bytes.

The first step in the gap test finds cache capacity and associativity. The test uses the reference strings to conduct a series of parameter sweeps over n, k, and o, organized as shown in the pseudocode in Figure 2.1. It measures the time taken for many repetitions of the reference string and conducts a simple analysis on those results. The test takes four inputs: a lower bound on cache size, *LB*; an upper bound on cache size, *UB*[3]; an upper bound on associativity, *MaxAssoc*, and the operating system page-size, available from the POSIX `sysconf` interface.

The intuition behind this part of the algorithm is simple. Consider a direct-mapped cache. The algorithm first tries the set of reference strings from *G(2,LB,0)* to *(2,UB,0)*. When k reaches the L1 cache size, the two locations in the reference string will map to the same cache location and each reference will miss in the L1 cache. That effect increases t above *baseline* and causes the conditional in the loop

[3]The upper bound on the cache size is necessary to allow the benchmark to terminate when the system does not have a cache.

to test positive, terminating the first loop nest and recording associativity of one and the correct cache size.

With a set associative cache, the sweep will continue until n is one greater than the associativity and $(n-1) \cdot k$ equals the cache capacity. At that point, the locations in the reference string all map to the same set and, because there are more references than ways in the set, the references will begin to miss. For smaller values of n, all the references will hit in cache and the time will match the baseline time.

In the second step, the algorithm uses the same effect to find line size. It already has values for n and k that match capacity and associativity. It runs a parameter sweep on o in the reference string *G(n,k,o)*. When o, the offset in the last block, reaches the line size, the last access in the string maps into a different set in cache, all n references hit in cache, and the measured time returns to *baseline*.

Of course, both steps assume that we can measure the running time of the reference string with a high degree of precision and that the compulsory misses at the start of the reference string do not matter.

Running a Reference String To measure the time for a reference string, the implementation must instantiate the string and walk its references enough times to obtain an accurate timing. For the tests described in this chapter, we instantiate the reference string by creating an array of pointers (in C, we use an array of type `void **`) and create a circular linked list that includes each location once.

The loop that runs the reference string is simple:

```
loads ← number of accesses
start ← timer()
while (loads- > 0)
  p ← *p;
finish ← timer()
elapsed ← finish - start
```

The implementation unrolls the loop by a factor of ten to ensure that the loop overhead is small relative to the memory access costs. We select a number of accesses that is large enough to ensure that the fastest test, *G(2,LB,0)*, runs for at least 1,000 timer ticks.

Timing a Reference String The pseudocode to run a reference string computes elapsed time using a set of calipers, the calls to *timer*, placed immediately outside a minimal timing loop. In practice, obtaining good times is difficult. Our task is made more difficult by the need to run on arbitrary POSIX systems in ordinary multiuser mode (e.g., not in single-user mode). To obtain accurate timings in this environment, we use a simple but rigorous discipline.

First, we use an accurate timer. The timer uses the POSIX `gettimeofday` call and combines the `tv_sec` and `tc_usec` fields of the `timeval` that it returns to produce a double-precision floating-point value. We scale the number of accesses to the apparent resolution of this timer, determined experimentally.

Second, we run many trials of each reference string to find the minimum measured execution time. We want the shortest time for a given reference string; outside interference manifests itself in longer times. To find the shortest time, we adopt a threshold value, *Trials*, and run the test repeatedly until we have not seen a change in the minimum execution time in the last *Trials* runs.

Finally, we convert the measured times into cycles. We carefully measure the time required for an integer add and convert the measured time into integer-add equivalent units. Specifically, we multiply to obtain nanoseconds, divide by the number of accesses, and round the result to an integral number of integer-add equivalents. This conversion eliminates the variability introduced by amortized compulsory misses and loop overhead.

Experimental validation on a broad variety of machines shows that these techniques produce accurate results for their L1 cache characteristics (see Section 2.4).

We use the same basic techniques, applied with different reference strings, in our other tests.

Reducing the Running Time Figure 2.1 suggests that the parameter sweeps sample the space at a fine and uniform grain. The implementor can radically improve the running time by reducing the number of sampled points. On most systems, for example, the size of the gap, k, will be an integral multiple of 1 KB. Associativity is unlikely to be odd. Line size is likely to be a power of two. The current implementation uses $LB = 1\,\text{KB}$, $UB = 16\,\text{MB}$, and an initial 1 KB increment that increases in steps as k grows. It tests n for the values 2 and odd numbers from 3 to 33. It varies o over powers of two from `sizeof(int)` to page size.

Limitations The gap test only works if it can detect the actual hardware boundary of the cache. We do not apply the gap test beyond L1 for several reasons. Higher levels of cache tend to be shared, either between I-cache and D-cache, or between cores, or both. Operating systems lock page table entries into higher-level caches. Higher levels of cache often use physical rather than virtual addresses. Any of these factors can cause the gap test to fail. It works on L1 because L1 caches are core-private and virtually mapped, and page tables are locked into L2 or L3 cache.

The gap test, as we have formulated it, is similar to the test used in X-RAY [51]. Our improvements for this test lie in the techniques for measuring and filtering the results.

2.2.2 Cache-Only Test

The second test that our tools apply is a more general test for cache capacity. This test isolates cache effects from TLB effects. Like the gap test, it performs a sweep over a parameterized reference string. It uses the same infrastructure as the gap test to run and time the string.

The reference string in the cache-only test, *C(k)*, is designed to minimize the

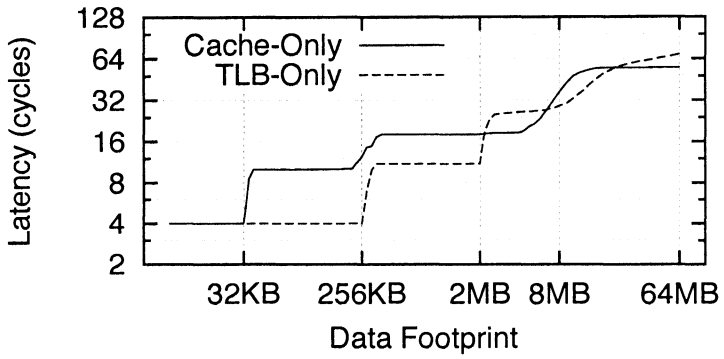


Figure 2.2: Intel E5530 response, log-log plot

impact of TLB misses. The parameter k specifies the reference string's memory footprint. The generator for *C(k)* knows both the L1 line size, from the gap test, and the operating system page-size.

Given k, the L1 line size, and the page size, the generator builds an array of pointers that uses k bytes of memory. The generator constructs an index set, the column set, that covers one page and accesses one pointer in each line on the page. It constructs an index set, the row set, that contains the starting address of each page in the array. It shuffles both the column and row sets into random order.

To build the linked list, it iterates over the row set, choosing pages. Within each page, it links together the lines on that page in the order specified by the column set. It links the last access in one page to the first access in the next page. If page size does not divide k, it generates a partial last row in random order.

To measure cache capacity, the test uses this reference string in a simple parameter sweep:

for $k \leftarrow$ *LB to UB*

  $t_k \leftarrow$ *time for C(k)*

The implementation, of course, is more complex. It uses the methods described for the gap test to run and time the reference string. The sweep produces a series of values,

t_k, that form a piecewise linear function describing the processor's cache response.

The *cache only* line in Figure 2.2 shows the result of running the cache-only test on an Intel Nehalem E5530 processor. Notice the soft transitions on the L2 and L3 caches. These reflect the behavior an actual program sees from this 256 KB, unified I and D L2 cache. Our analysis reports an effective L2 capacity of 224 KB from this dataset.

As long as page size is large relative to line size, *C(k)* produces clean results that isolate the cache behavior. To draw conclusions from the data, however, requires analytical techniques explained in Section 2.3.

2.2.3 TLB Test

The third test that our tools apply attempts to isolate TLB behavior. The TLB test builds on the same ideas as the cache-only test. It constructs a reference string that isolates TLB behavior and performs a parameter sweep over the total size covered by the reference string. The parameter sweep produces a piecewise linear function that must be subjected to further analysis.

To isolate TLB behavior, the test generates a reference string *T(n,k)*. For a given n and k, *T(n,k)* access n pointers in each page of an array of k bytes. To construct *T(1,k)*, the generator builds a column index set and a row index set as in the cache only test. It shuffles both sets. To generate the permutation, it iterates over the row set choosing pages. It chooses a single line within the page by using successive lines from the column set, wrapping around in a modular fashion if necessary. The result is a string that accesses one line per page, and spreads the lines over the associative sets in the lower level caches.

For $n > 1$, the generator uses n lines per page, with a variable offset within the page to distribute the accesses across different sets in the caches and minimize associativity conflicts. The generator randomizes the full set of references, both to avoid the effects

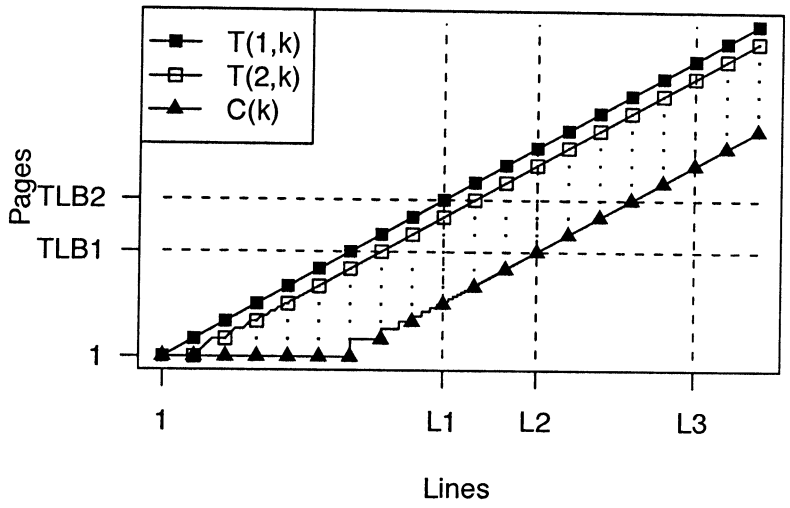


Figure 2.3: Memory Hierarchy Search Space

of hardware prefetch and to avoid successive accesses to the same page.

To measure TLB response, the test uses this reference string in a simple parameter sweep:

for $k \leftarrow$ *LB to UB*

$t_k \leftarrow$ *time for T(1,k)*

The parameter sweep uses the same infrastructure as the gap test and the cache-only test. The *tlb-only* line in Figure 2.2 shows the result of running the TLB test on an Intel Nehalem E5530 processor. For the TLB data, the x-axis represents total footprint covered by the TLB, or *pages* $\times$ *page size.*

Eliminating False Positives The cache-only test hides the impact of TLB misses by amortizing those misses over many accesses. Unfortunately, the TLB test cannot completely hide the impact of cache. The underlying reason is simple: any action that amortizes cache misses will also partially amortize TLB misses. To see this, consider the plot in Figure 2.3. This log-log plot depicts the set of feasible memory-footprints that we can test. The x-axis shows the number of lines in a given footprint, while the y-axis shows the number of pages. Labeled lines show boundaries of cache and TLB

levels.

Consider the footprint of the cache-only string, *C(k)*, as k runs from one to large. *C(1)* generates the footprint (1,1) in the plot. *C(2)* generates (1,2), and so on. When k reaches $\frac{\text{page size}}{\text{line size}}$, it jumps from one page to two pages. *C(k)* forms a step function that degenerates to a line due to the log-log form of the plot. In contrast, the TLB string, *T(1,k)*, has a footprint that rises diagonally, at one page per line.

The plot predicts points where performance might change. When the line for a given reference string crosses a cache or TLB boundary in the memory hierarchy, performance may jump. With *C(k)*. we see a jump when it crosses the cache boundaries but not when it crosses the TLB boundaries—precisely because the order of access amortizes out the TLB behavior. If we take the references in *C(k)* and randomize them, the TLB misses should become more visible in the results. Of course, if the hardware responds with a rise in access time before the actual boundary, the test shows that point as the effective boundary.

When the TLB line crosses a cache boundary, the rise in measured time is indistinguishable from the response to crossing a TLB boundary. The plot, however, gives us an insight that allows us to rule out false positive results. The line for *T(2,k)* parallels the line for *T(1,k)*, but is shifted to the right. If *T(1,k)* shows a TLB response at x pages, the *T(2,k)* shows a TLB response at x pages. Because *T(2,k)* uses twice as many lines at x pages as *T(1,k)*, a false positive response caused by the cache in *T(1,k)* will appear at a smaller size in *T(2,k)*.

To detect false positives, the TLB test runs both the *T(1,k)* and *T(2,k)* reference strings. It performs the complete analysis on the resulting data, which produces a list of TLB suspect points, in ascending order by k. If *T(1,k)* shows a rise at some point, say x pages, then one of several cases can occur. If the rise is due to a TLB, a corresponding rise will occur at x pages in *T(2,k)*. On the other hand, if the rise is due to some cache effect, *T(2,k)* will occur with the same number of lines but a

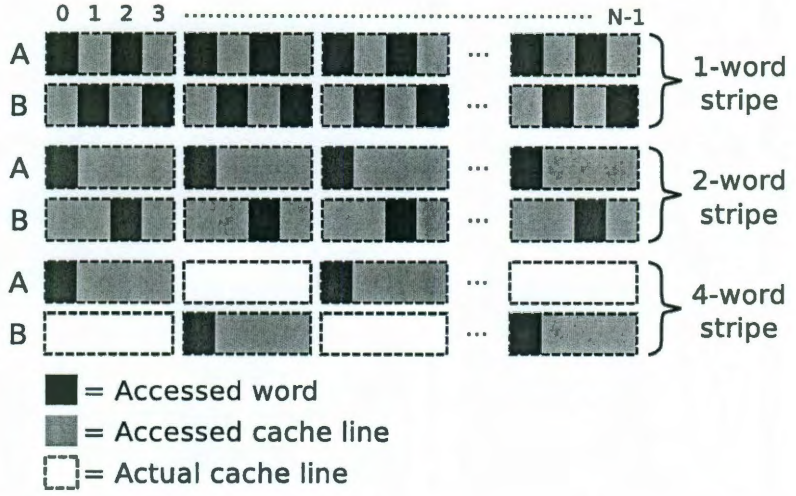


Figure 2.4: Line size micro-benchmark access pattern

different number of pages. Comparison of the lists of suspect points from the *T(1,k)* and *T(2,k)* reference strings exposes most false positive results.

Unfortunately, a worst-case choice of cache and TLB sizes can fool this test. If *T(1,k)* maps into m cache lines at x pages, and *T(2,k)* maps into $2 \cdot m$ cache lines at x pages, and the processor has caches with m and $2 \cdot m$ lines, both reference strings will discover a suspect point at x pages. The analysis will report a TLB boundary at x pages. Using more tests, *e.g.*, *T(3,k)* and *T(4,k)*, could rule out these points. In practice, cache sizes tend to grow by more than a factor of two, so we expect this case to be uncommon.

2.2.4 Line Size

Given effective cache capacity for each level, our tool next finds the line size at each level. As a practical necessity, the line-size test cannot rely on associativity for two reasons. First, the cache-only test discovers effective cache sizes that do not trigger the obvious associativity effects. Second, physically-mapped caches also disrupt the associativity behavior. Thus, the line-size test relies on effects directly related to line size: spatial locality and conflict misses.

The test generates a reference string *L(n,s)*, where n is the cache capacity and s is the stripe, or line size to test. For each cache level of size n the test performs a parameter sweep over *L(n,s)* for *sizeof(void*)* $\leq s \leq$ *page size/2*. To save time we limit s to values that are powers of two, but the test works for any s within the given bounds.

The test *L(n,s)* generates two complementary striped access patterns, A and B, depicted in Figure 2.4. Pattern A accesses the first location in each of the even numbered stripes while pattern B accesses the first location in each of the odd numbered stripes. The value of s determines the width of each stripe. Both patterns are constructed to span the entire cache capacity, resulting in a combined span of twice the cache capacity; that is, the number of pages referenced by both patterns is twice the number of pages that the cache can hold. But, because each pattern only accesses half of the stripes, the total data footprint is no larger than the cache capacity. In other words, the total number of bytes accessed by both patterns is less than the number of bytes that the cache can hold. The test proceeds to access every location in pattern A followed by every location in B, repeating until sufficient timing granularity has elapsed. The accesses within each pattern are shuffled to defeat the prefetcher.

When patterns A and B both map to the same cache lines, they conflict. For any value of s smaller than the line size, each access generates a miss. This contention occurs because when the stripe is smaller than the line size, each pattern effectively accesses every cache line in the span of its pattern. Since the combined patterns span twice the cache capacity, the total number of lines accessed exceeds the cache capacity by a factor of two. However, when s is an integral multiple of the actual line size, patterns A and B no longer conflict. Intuitively, each pattern will leave empty "holes" into which the other pattern can fit. The test starts with a small value of s and increases it until A and B do not conflict, at which point the cost of running the reference string drops dramatically because the conflict misses disappear.

Consider the one-word stripe at the top of Figure 2.4. Since the actual line size in this example is four words, A and B access conflicting cache lines. Both patterns access two words per line, so the reference string achieves some spatial reuse. The test uses the latency measured with the one-word stripe as its baseline. With $s = 2$, A and B still conflict, but spatial locality decreases. With $s = 4$, A and B map to different lines, so conflict misses disappear completely and the time to run the reference string drops dramatically.

The analysis portion of this test is straightforward. Measured latency increases relative to the baseline as s increases due to the decrease in spatial locality. As soon as the stripe width is large enough to prevent conflict misses, measured latency drops below the baseline. The effective line size, then, is equal to the s for which the latency of *L(n,s)* is less than the latency of *L(n,sizeof(void*))*.[4]

For the line-size test to function properly both patterns A and B must map to the same cache lines. On a virtually mapped cache we can just create two adjacent arrays for A and B, both of length n. However, physically mapped caches do not guarantee that the arrays map contiguously into the cache. Our key insight is that physically mapped caches provide contiguous mapping *within* each page.

To leverage this observation, the test generates the access patterns at a *page size* granularity. It allocates *2*n/pagesize* pages and randomly fills half of them with pattern A and half with pattern B. Because the reference string spans twice as many pages as should fit in cache, on average *2*associativity* pages will map to each cache set, where *associativity* is the number of associativity sets for that particular level of cache.

Two competing pages can occupy the cache simultaneously if and only if two conditions are met: (1) one page contains pattern A and other page contains pattern B and (2) the stripe width is an integral multiple of the effective line size. Otherwise,

[4]A system with line size equal to wordsize produces the same response for all values of s. We have not encountered such a system.

the two pages conflict with each another. (Note that it suffices to have some, but not all, pages meet condition (1).)

We cannot, in a portable way, control the page mapping. We can, however, draw random samples from a large set of pages and mappings to look for these conditions. Our methodology for running a reference string achieves this effect. If s is smaller than line size, then condition (2) never holds and the measured latency remains high. For s equal to line size (or an integral multiple of line size), condition (2) always holds and condition (1) holds in some random samples. As long as the test runs enough trials, it will find the desired mapping in some samples and our timing methodology will record that time.

2.2.5 Associativity

Following X-RAY, our gap test detects associativity in the L1 cache, provided that it is virtually mapped [51]. X-RAY tests associativity in higher levels of cache by requesting superpages that are larger than the cache size, which forces the contiguous mapping that the test needs. Because superpage support is not yet portable, we did not follow that path; our tools do not measure associativity for caches above L1. More importantly, it is not clear that the compiler can rely on associativity effects in caches that have physical address mappings or that have an effective capacity smaller than the actual hardware capacity. In other words, if we cannot devise a portable test to measure a particular characteristic, then it seems unrealistic for a compiler optimization to be able to take advantage of that characteristic.

We have developed a straightforward test for TLB associativity based on the gap test. It functions well in most cases, but it can be fooled. One version of the ARM processor, in particular, has a two-part TLB structured as an 8-page, fully-associative TLB and a 56-page, 2-way set associative TLB. The lookup mechanism consults the small TLB first; a miss in the small TLB faults to the larger TLB. The TLB capacity

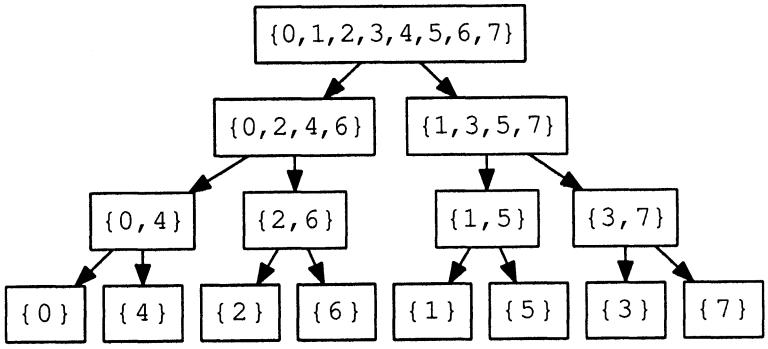


Figure 2.5: Cache-oblivious access pattern

test finds both the 8-page and the 56-page TLB. The associativity test reports that both TLBs are 8-way set associative; we have not been able to devise a reference string that exposes the 2-way associativity in the larger TLB.[5]

2.2.6 Cache Oblivious Access Order

Although the hardware prefetcher can be defeated by uniformly randomizing an access pattern, doing so may produce unexpected results if spatial locality is not considered. For example, consider a dense data footprint in which every contiguous word is accessed: if we randomize the access order uniformly, it is possible that two spatially adjacent words are accessed with temporal locality. If those two words map to the same cache line then only one of them will suffer a miss, potentially disrupting the expected performance result. If the line size is known then the access pattern can certainly be adjusted to avoid this situation by explicitly accessing only a single word per line. However, if the line size is unavailable (for example, if the gap test fails), then a biased random order can be used to eliminate benefits from spatial locality. Figure 2.5 illustrates our *cache-oblivious* access pattern that minimizes spatial locality without knowledge of any cache parameters. This concept is modeled after

[5]The fact that we cannot, in portable C code, discover the associativity suggests that the architects made a good decision. They used a smaller and presumably cheaper associativity precisely in a place where the compiler could not use the larger associativity.

cache-oblivious algorithms [25], although our goal is to minimize locality instead of to maximize it. The data is partitioned with a recursive divide-and-conquer strategy, just as with previous cache-oblivious algorithms; but instead of splitting into left and right halves at each level, the data is split into interleaving, strided sub-arrays. The leaves of the tree represent the access order from left to right; note that spatial locality is minimized as adjacent words in the original order are accessed far apart. The illustration still shows significant regularity in the final access pattern. To reduce the regularity of the pattern, we can increase the number of children at each level and randomize the order in which the children are visited. This preserves the desired spatial locality property while introducing randomization that is sufficient to defeat the prefetcher.

2.3 Automatic Analysis

The previous section focused on explaining the mechanics of the micro-benchmarks and only suggested informally how to interpret the results. This section describes our multi-step analysis technique for the cache and TLB capacity benchmarks. Each micro-benchmark produces a two-dimensional set of data points that map a data footprint size to an access latency, as shown in Figure 2.2. The data sets are plotted and analyzed with a base-2 logarithmic scale on both the capacity axis and the latency axis.

The goal of the analysis is derive two key pieces of information from that set of points: (1) the number of levels of cache or TLB and (2) the transition point between each level (*i.e.*, the capacity of each level). We discuss the algorithm in terms of cache capacity, but we use the same techniques to derive TLB capacity from the results of the TLB benchmark.

We designed our analysis algorithm to satisfy several important requirements.

The analysis is *fully automatic*; allowing human intervention would obviously contradict the goal of automatically characterizing a system. More importantly, manual interpretation of micro-benchmark results is not something that a normal developer or scientist can be expected to understand. Even trained experts may make different judgment calls when faced with a tradeoff; thus, the analysis should be deterministic and *objective*. We define an objective function that quantifies the goodness of a particular interpretation, allowing the analysis to find an optimal solution using mathematical optimization techniques. The analysis is *conservative*. In the presence of ambiguous results the analysis will favor underestimating that cache capacity rather than overestimating it, ensuring that program transformations will not overutilize the cache. Finally, the analysis is *robust*. In addition to the experimental validation in Section 2.4, we provide clear justification for our analysis and avoid using arbitrary thresholds. Although we cannot prove that our analysis will always draw perfect conclusions in the presence of noisy or unexpected results, our thorough testing and analytical justifications certainly increase our confidence that it will at least produce reasonable answers.

The following sections describe the three steps of our analysis: (1) filtering noise, (2) determining the number of levels of cache and (3) determining the capacity of each cache level.

2.3.1 Filtering Timing Noise

Because our micro-benchmarks measure actual system behavior, timing error is a major obstacle to correctly interpreting the micro-benchmark results. Ideally, each micro-benchmark should execute on an otherwise idle system. We have no portable way to request single-user mode or real-time priority; thus, the timing results are likely to reflect transient system events of the OS or other daemon processes. Our tools use a two pronged approach to minimize timing error: we reduce such errors

during collection and we filter the data after collection to remove any remaining noise.

Our timing methodology, introduced in Section 2.2.1, provides the first-line defense against timing error. The tests perform multiple trials for each value in the parameter sweep, but only keep the best (*i.e.*, the minimum). To prevent transient system events from affecting multiple trials of the same parameter value, we sweep across the entire parameter space before repeating for the next trial. Thus, an anomaly introduced by system activity is spread across one trial at several parameter values rather than multiple trials at the same value. As mentioned earlier, we repeat the trial at each parameter value until we find *Trials* consecutive attempts with no improvement in the minimum value. In practice, the tests use a value of 100 for *Trials*. This adaptive approach collects more samples when the timing results are unstable and fewer samples when the results are consistent. It always collects at least *Trials* samples per point.

During post-processing we filter the data to remove any remaining noise. Our filtering scheme leverages two observations. First, we assume that cache latency is an integral number of cycles, so we divide the empirical latency by the cycle duration and round to the nearest integer.[6] For the sizes that fit in a cache, all accesses should be hits and should, therefore, take an integral number of cycles. For sizes that include some misses, the total latency is a mix of hits and misses. Rounding to cycles in these transitional regions produces a slight inaccuracy, but one that has minimal impact on the analysis. As the data approaches the next cache boundary, all the references are misses in the lower level cache and the latency is, once again, accurate.

Second, we assume that the latency curve that the empirical results approximate is *isotonic*, or non-decreasing. That is, we don't expect the latency to decrease when data footprint increases. Sometimes, the empirical results contain non-isotonic data points. We correct these anomalies with a statistical technique called *isotone re-*

[6]Colleagues developed a separate micro-benchmark that portably determines the time required to perform a single integer add. We use that value as a proxy for cycle time.

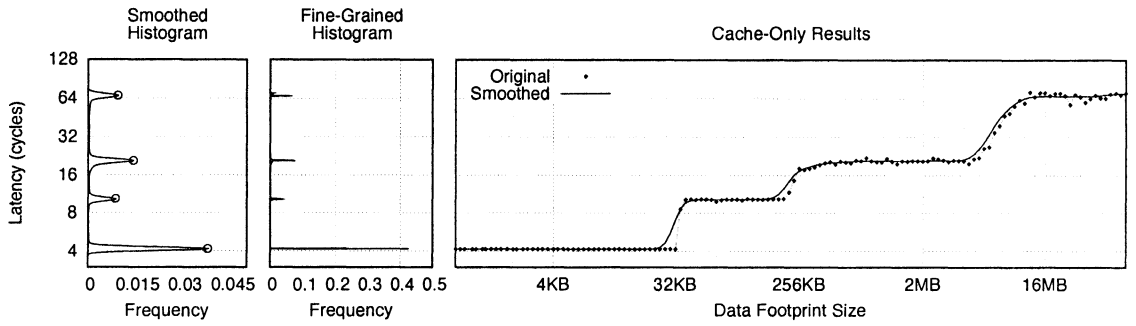


Figure 2.6: Histogram Analysis for Intel Xeon E5530 Nehalem

gression, which removes decreasing regions from a curve with a form of weighted averaging. We use the Pool Adjacent Violators Algorithm as described by Robertson *et al.* [42].

2.3.2 Determining the Number of Cache Levels

The next step in analysis is determining the number of levels in the cache hierarchy. Because this step only determines the rough global structure of the curve, it can use aggressive smoothing techniques, as long as they preserve the curve's important features. In the final step of the analysis, finding the transition points between levels, such aggressive smoothing cannot be used because it may blur the precise transition points.

To begin, we apply a Gaussian filter to smooth the curve. The Gaussian filter serves as a low-pass filter that eliminates noise while preserving the curve's global shape. We use a filter window whose width is derived from the minimum distance that we expect between two cache levels. We assume that each cache level should be at least twice as large as the previous level; on a log_2 scale the appropriate window width is $log_2(2) = 1$. With this window, the filter aggressively smoothes out noise between cache levels but cannot filter out an actual level unless it is less than twice the size of the previous level. The smoothed curve in the rightmost graph in Figure 2.6

shows the results of a Gaussian filter applied to the original data points.

Next, we identify regions in the curve that correspond to individual levels in the cache. Informally, we want to find relatively flat regions of the curve that are surrounded by sloped regions. We can detect such regions by computing a one-dimensional density estimate along the y-axis. We estimate the density with a fine-grained histogram of the y-values. This technique splits the y-axis into a large number of adjacent bins and computes the number of points that fall in the y-range of each bin. Intuitively, the bins that contain flat regions will have much larger counts than bins that contain sloped regions. Thus, a cache level should be marked by a region of high density surrounded by regions of low density.

The fine-grained histogram, shown rotated sideways in Figure 2.6 to match the y-axis of the cache-only plot, provides a rough indication of the desired information. Further smoothing clarifies the region structure. As before, we apply a Gaussian filter. This time, however, the filter window width derives from the minimum expected magnitude of a transition between regions—that is, the minimum relative cost of a cache miss. We assume that a cache miss incurs at least a 25% performance penalty; this step of the analysis considers anything less to be insignificant. That assumption implies that the window width, on a log_2 scale, should be $log_2(1.25) \approx 0.322$.

With this filter window width, the Gaussian filter consolidates the adjacent bins and produces a smooth curve with clear maxima and minima. The leftmost graph in Figure 2.6 depicts the smoothed histogram. The final step counts the number of local maxima in the curve. We compute the discrete first derivative (*i.e.*, the slope) of the smoothed histogram. Local maxima correspond to points where the first derivative changes from non-negative to negative. This simple algorithm detects the peaks in the histogram, as indicated by the circles on the peaks of the smoothed histogram in Figure 2.6. Each peak corresponds to a distinct level in the memory hierarchy. If the analysis finds n peaks, that indicates $n - 1$ levels of cache, plus main memory. This

step of the analysis concludes by returning the number of levels in the cache.

2.3.3 Determining the Size of Each Cache Level

The final step in analysis identifies the transition points between levels in a cache curve. These points correspond to the footprint sizes where access latency begins to rise because a level of cache is effectively full. This section presents an intuitive algorithm to find objectively the optimal points to split a curve, given the number of levels in the cache.

Interpreting the cache-latency curve is somewhat subjective, as it entails a judgment call with regard to the capacity/latency tradeoff. The ideal curve would resemble a step function, with long, flat regions connected by short steep transitions. On such a curve, cache capacity is easily determined as the final point before the rise in latency. However, modern processors show soft response curves that rise considerably before the hardware cache boundary, at least on the higher levels of cache. Some previous approaches try to estimate actual cache capacity based on the shape of the latency curve. We are, instead, interested in deriving a number that makes sense for compiler-based blocking of memory accesses. That number, the *effective cache capacity*, corresponds to the capacity at which access latency starts to increase.

Using larger numbers to block memory accesses makes little sense. The performance of a loop blocked to hardware cache capacity may be significantly worse than one blocked to the effective cache capacity. Thus, our analysis tries to identify the last point at which performance remains "flat". Even so, "flat" can be quite subjective if the transition begins with a very gradual slope. In some cases, the effective cache size can be extended for a relatively minor latency penalty. At some point, however, the performance cost of extending the effective capacity is not worth the extra capacity. Our experience suggests that even human experts may interpret these graphs differently. In the face of such ambiguity, it is important to choose an objective policy

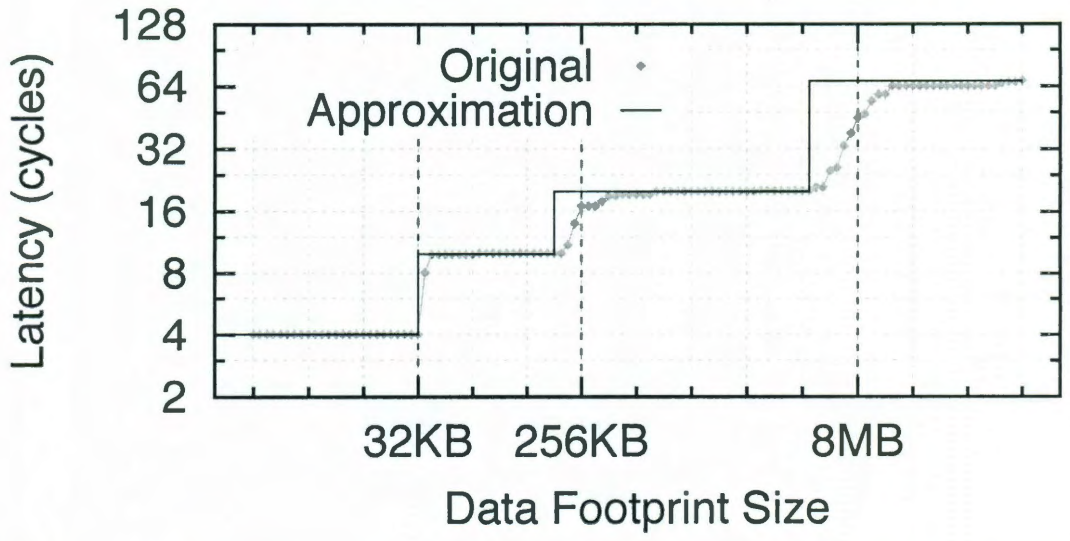


Figure 2.7: Step-Function Approximation for Intel Xeon E5530 Nehalem

and trust the results of the analysis. In keeping with the principle that compilers make conservative decisions in the face of ambiguity, we have opted for an objective function that selects effective cache capacities that occur early in the transition region of the curve. The rest of this section formalizes our objective function and the optimization algorithm that solves it.

Cache blocking algorithms attempt to fully use each cache level, expecting that the worst-case latency for each level is similar to its average latency. (Otherwise a performance benefit may arise from not using the entire cache). Consequently, our interpretation should summarize a region of the curve as the worst-case latency for that region. This decision leads intuitively to a step-function that is flat across the effective region of the cache and takes its upward step at the transition point between two levels. Thus, we approximate the cache-latency curve with a step function. The number of steps in the approximation should equal the number of levels in the cache, determined by the earlier analysis. We define, then, an objective function that is the error between the step-function approximation and the original cache curve.

Given this objective function, we employ a simple dynamic programming algo-

rithm to find the optimal split points that minimize the objective function. Our algorithm is based a polygonal approximation algorithm by Perez *et al.* [40]; we extend the algorithm to compute a step-function approximation instead of a polygonal approximation. Although the theoretical time complexity of this algorithm is $\Theta(MN^3)$, where M is the number of levels of cache and N is the number of data points, the running time is not a practical problem. The values for M and N are small enough that the cost of running the analysis is insignificant relative to the cost of running the micro-benchmark to gather the data.

Figure 2.7 shows the result of the step-function approximation. Notice that the original curve has not been processed by the Gaussian filter, because doing so would alter the transition points. The post-processing described in Section 2.3.1 is still helpful, however. The first three horizontal steps in the approximation summarize the first three levels of cache. The right endpoint of a step indicates the size of that level of cache while the height of the step determines its worst-case access latency. Although the L2 and L3 transitions are gradual in this example, the approximation conservatively identifies the beginning of the slope as the effective cache size. If the slope were more gentle, the global optimization algorithm may have selected a larger effective cache size with a slightly longer access latency. In any case, transition points are always selected to minimize the error of the step-function approximation.

Finally, the rightmost step in the approximation corresponds to main memory, indicating the penalty for missing in the last level of cache. The right endpoint of the main-memory step is theoretically bounded by the size of the system's main memory, but our benchmark tests footprints only as large as the upper bound, *UB*, specified in Section 2.2.1.

2.4 Experimental Validation

To validate our ideas and our tools, we run them on a collection of systems that range from commodity x86 processors through an IBM Power 7, an ARM, and the IBM Cell processor in a Sony Playstation 3. All of these systems run some flavor of Unix; all of them support enough of the POSIX interface to run the tools.

Table 2.1 shows the measured cache parameters: line size, associativity, capacity, and latency for each level of cache that the tools detect. The *Measured* column shows the numbers produced by the tools. A blank in one of the *Measured* columns means that the tools do not measure that value (e.g., associativity on an upper level cache). The *Actual* column lists the actual number for that processor, if we have been able to find documentation describing it. Table 2.2 shows the capacity numbers for TLBs on the same systems. We do not show page-size numbers in the table; they are available from `sysconf`.

The tables are produced by a semi-automatic process. We have a script that distributes the code, uses the Unix `make` utility to build and execute them, and retrieves the results. Four of the systems use batch queues; those systems require manual intervention to schedule the job and retrieve the results.

Figures 2.8 and 2.9 show visual representations of the data-cache and TLB capacity results. A horizontal bar depicts each system's physical cache or TLB hierarchy, but the bar is normalized so that each level corresponds to the appropriate label on the x-axis. This normalization is linear between the major x-axis tic marks, but non-linear across the entire x-axis. The tic marks that appear superimposed on top of each bar correspond to the effective cache or TLB capacities returned by the benchmark. This plot allows easy evaluation of the effectiveness of the benchmark by observing where the effective capacities fall in comparison to the physical capacities.

A couple of entries deserve specific attention. The Power 7 has an unusual L3 cache structure. Eight cores share a 32 MB L3 cache, but each core has a 4 MB portion

Processor		Line size (b)		Assoc.		Capacity (kb)		Lat. (cyc.)
		Act.	*Exp.*	*Act.*	*Exp.*	*Act.*	*Exp.*	*Exp.*
AMD Opteron 2360 SE Barcelona	1	64	64	2	2	64	64	3
	2	64	64	16		512	448	12
	3	64	64	32		2048	1792	46
AMD Opteron 275	1	64	64	2	2	64	64	3
	2	64	64	16		1024	896	17
AMD Opteron 6168 Magny-Cours	1	64	64	2	2	64	64	3
	2	64	64			512	512	13
	3	64	64			12288	5120	32
AMD Phenom 9750 Agena	1	64	64	2	2	64	64	3
	2	64	64	16		512	448	12
	3	64	64	32		2048	2048	31
ARM926EJ-S	1	32	32	4	4	16	16	2
	2	32	32	?		256	224	15
IBM Cell (PS3)	1	128	128	?	4	32	32	2
	2	128	128	?		512	320	20
IBM POWER7	1	128	128	8	8	32	32	1
	2	128	128	8		256	256	6
	3	128	256	?		32768	3072	15
	4		256				20480	51
Intel Core 2 Duo T5600 Merom	1	64	64	8	8	32	32	3
	2	64	128	8		2048	1280	14
Intel Itanium 2 900 McKinley	1	64	64	4	4	16	16	2
	2	128	128			256	256	6
	3	128	128			1536	1024	18
Intel Itanium 2 9040 Montecito	1	64	64	4	4	16	16	2
	2	128	128	8		256	256	6
	3	128	128	12		12288	4096	11
Intel Pentium 4	1	64	64	4	4	8	8	4
	2	64	128			512	256	36
Intel Xeon E5420 Harpertown	1	64	64	8	8	32	32	3
	2	64	128	24		6144	4096	15
Intel Xeon E5440 Harpertown	1	64	64	8	8	32	32	3
	2	64	64	24		6144	4096	15
Intel Xeon E5530 Nehalem	1	64	64	8	8	32	32	4
	2	64	64	8		256	224	10
	3	64	64	16		8192	5120	19
Intel Xeon E7330 Tigerton	1	64	64	8	8	32	32	3
	2	64	128	12		3072	1792	14
Intel Xeon X3220 Kentsfield	1	64	64	8	8	32	32	3
	2	64	64			4096	2560	15
Intel Xeon X5660 Westmere	1	64	64	8	8	32	32	4
	2	64	64	8		256	224	10
	3	64	64	16		12288	8192	22
PowerPC 7455 G4	1	32	32	8	8	32	32	3
	2	64	64	8		256	224	10
	3	128	128	8		2048	1536	32
PowerPC 750 G3	1	32	32	8	8	32	32	2
	2	128	128	2		1024	512	20
Sun UltraSPARC T1	1	16	16	4	4	8	8	4
	2	64	64	12		3072	3072	23

Table 2.1: Cache Results

Processor		**Capacity in KB**	
		Actual	*Measured*
AMD Opteron 2360 SE Barcelona	1	192	192
	2	2048	2048
AMD Opteron 275	1	128	128
	2	2048	2048
AMD Opteron 6168 Magny-Cours	1	192	192
	2	2048	2048
AMD Phenom 9750 Agena	1	192	192
	2	2048	2048
ARM926EJ-S	1	256	32
	2		224
IBM Cell (PS3)	1	?	256
	2	?	4096
IBM POWER7	1	4096	4096
	2	?	32768
Intel Core 2 Duo T5600 Merom	1	64	64
	2	1024	1024
Intel Itanium 2 900 McKinley	1	2048	7680
	2	8192	
Intel Itanium 2 9040 Montecito	1	512	1920
	2	2048	
Intel Pentium 4	1	256	256
Intel Xeon E5420 Harpertown	1	64	64
	2	1024	1024
Intel Xeon E5440 Harpertown	1	64	64
	2	1024	1024
Intel Xeon E5530 Nehalem	1	256	256
	2	2048	2048
Intel Xeon E7330 Tigerton	1	64	64
	2	1024	1024
Intel Xeon X3220 Kentsfield	1	64	64
	2	1024	1024
Intel Xeon X5660 Westmere	1	256	256
	2	2048	2048
PowerPC 7455 G4	1	512	512
	2		1280
PowerPC 750 G3	1	512	512
	2		1280
Sun UltraSPARC T1	1	512	3840

Table 2.2: TLB Results

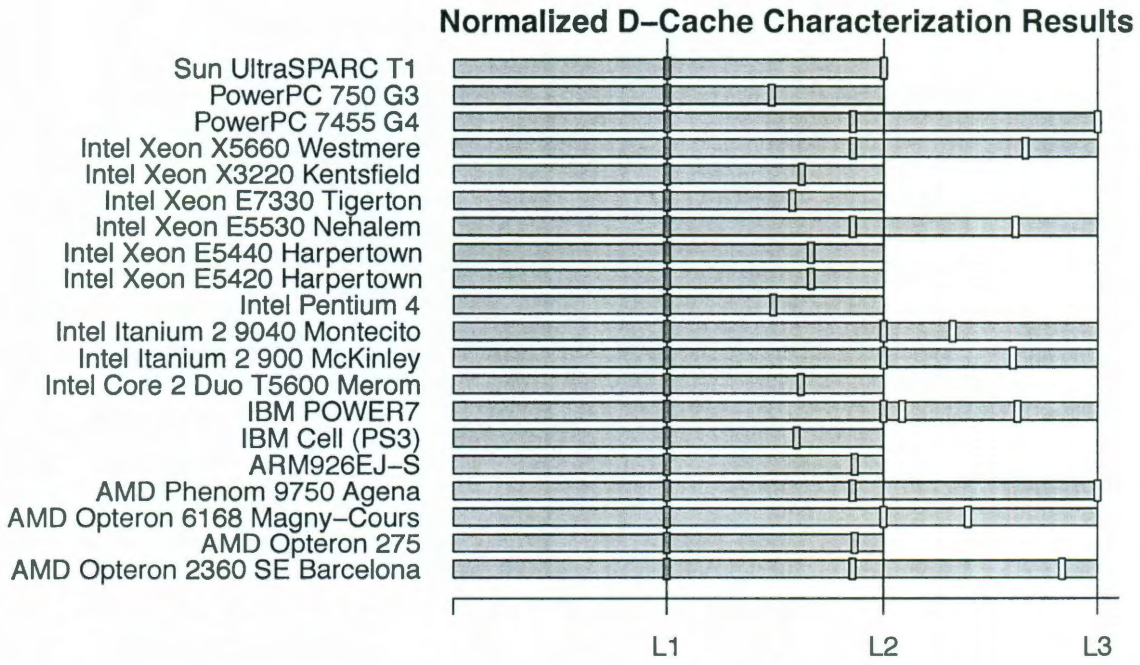


Figure 2.8: Normalized Data-Cache Capacity Results

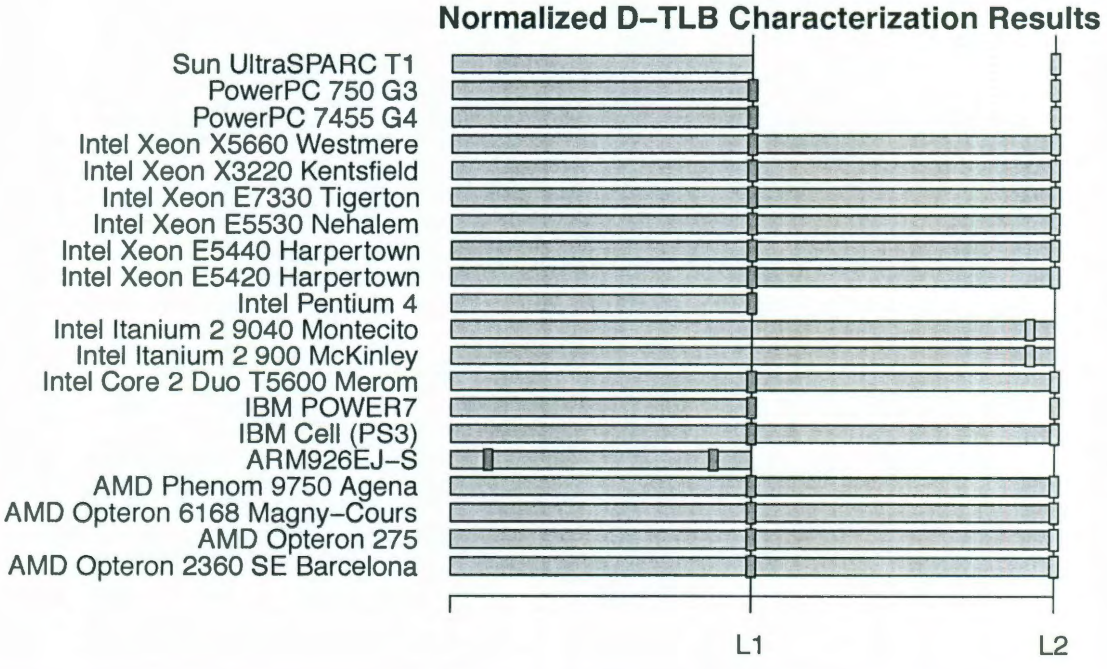


Figure 2.9: Normalized Data-TLB Capacity Results

of that cache that it can access more quickly than the remaining 28 MB. The tool discovers two latencies from this setup: a 2.5 MB cache with a 12 cycle latency and a larger 16 MBcache with a 35 cycle latency. Our tests were run on an active system; the effective sizes reflect the actual behavior that a program might see. A compiler that blocks for Power 7 caches would do better to use the tool's description than to treat it as a unified 32 MB L3 cache.

The TLB on the ARM system is another case where the tools produce results that differ from the hardware description. The manuals suggest that the ARM chip has one TLB that the program should see and that the TLB is implemented in two pieces: an 8 page fully associative TLB and a 56 page 2-way set associative TLB. The hardware first consults the small TLB; a miss in the small TLB triggers a lookup in the larger TLB. The tools discover this structure, but the sequential nature of the access causes the tool to classify the TLB as two levels.

Several of our experimental platforms exhibit cache designs that use different line sizes for different levels of cache. The Intel Itanium, PowerPC G3, and Sun T1 all use a smaller line size for L1 and a larger line size for higher levels of the cache. The PowerPC G4 actually uses a different line size for each level of cache. Our test correctly detects the line sizes for each level of cache on these systems. Additionally, on several systems we detect a larger *effective* line size for the last level of cache. These systems include the IBM POWER7, the Intel Core 2 Duo T5600 Merom, the Intel Pentium 4, the Intel Xeon E5420 Harpertown, and the Intel Xeon E7330 Tigerton. While it is possible that the documentation is incorrect for these systems, it seems much more likely that the effect is caused by a hardware prefetcher or the design of the memory controller. These examples reinforce the necessity to experimentally determine such characteristics rather than rely on documentation.

2.4.1 Effective Cache Sizes

As stated earlier, the benchmarks measure the effective cache sizes rather than the actual hardware cache size. The results show that effective size is typically much smaller than the actual size. For the L2 cache and above, the effective size can be as small as 50–75% of actual size. On L1 caches, the effective size usually matches the actual size.

The main reason for the difference between effective and actual cache sizes appears to be the use of physical mapping in the higher level caches. The data footprint of the test's reference string spans a contiguous virtual address space, but when translated to physical addresses it is no longer guaranteed to be contiguous. Because we have no portable way to control the virtual-to-physical mapping, we must accept the fact that the reference string maps pseudo-randomly into cache. When the reference string may not map efficiently into cache, because the virtual-to-physical map overfills some of the associativity sets, it produces a smaller effective capacity.

The problems with small effective cache sizes in physically mapped caches led González-Domínguez *et al.* to employ a probabilistic binomial model to estimate the actual cache size based on behavior of a large number of random samples [27]. We did not follow that approach for two reasons. First, we are interested in applying the cache capacity numbers to block programs for performance; in this situation, the effective sizes that we measure are better numbers for the compiler. Second, we believe that other factors than physical mapping contribute to reduced effective cache size. These include unified I-cache and D-cache hierarchies at the higher levels, cache sharing between cores, and MMU designs that cause page table entries to occupy cache space. From a practical perspective, however, we are less concerned with why the effective sizes are smaller than the actual sizes than we are with measuring the effective size, since that is the number the compiler needs to know.

2.5 Conclusions

This chapter has presented techniques that measure the effective sizes of levels in a processor's cache and TLB hierarchy, with the intent that these numbers be computed when the compiler is installed and used in each compilation to guide blocking for memory. The tools are portable; they rely on a C compiler and the POSIX operating system interfaces. The tools discover effective cache and TLB sizes that are suitable for use in optimizations that try to improve memory hierarchy behavior—in fact, these effective numbers should provide better optimization results than would be obtained using the actual hardware values from the manufacturer's manual.

Chapter 3

Instruction Cache Characterization

3.1 Introduction

The previous chapter presented micro-benchmarks for characterizing the cache and TLB hierarchy from the perspective of the processor's data access mechanism. Although most upper levels of the memory hierarchy are *unified* (*i.e.*, they contain both data and instructions), the data-centric memory benchmarks can ignore the impact of the instruction footprint because the driver loop in the main kernel is small (approximately 3 instructions before loop unrolling). Therefore, the data-centric benchmarks estimate the true effective capacity of each level of cache, regardless of whether a level is unified or only contains data. Yet, real applications usually exhibit larger instruction footprints than that of our benchmark—sometimes large enough to overflow instruction caches or introduce contention in unified caches. Thus, it is important for our micro-benchmark suite to also characterize the memory hierarchy from the perspective of a processor's instruction access mechanism. This chapter presents a micro-benchmark for characterizing the instruction memory hierarchy and determining which levels in the hierarchy are unified.

For convenience, we will refer to the instruction memory hierarchy simply as the

instruction cache, or I-CACHE, even though the hierarchy comprises both cache and TLB. Similarly, we will refer to the data memory hierarchy simply as the D-CACHE. We do not refer to a separate unified memory-hierarchy; instead, a level in the hierarchy is unified if and only if it appears in both the I-CACHE and D-CACHE hierarchies. Note that a unified level must be the same physical cache in both hierarchies, rather than two disjoint caches of equal size; our benchmark distinguishes between the two cases.

3.1.1 Motivation

There are several ways in which the compiler can use I-CACHE information for optimization. First, the compiler can limit code growth of any optimizations that affect code size (*e.g.*, loop unrolling or function inlining). This limit, based on actual machine behavior, would allow each optimization to maximize its benefit without incurring a performance penalty from overflowing the I-CACHE. Second, the compiler can ensure that data-cache transformations reserve enough space in unified caches for both the data and the instruction footprints. For example, knowing which levels of cache are unified allows the compiler to quantify the tradeoff between loop unrolling and loop tiling. Finally, if a program's actual instruction footprint only fills a fraction of the I-CACHE, the compiler can view this as an inefficient use of system resources. Essentially, the compiler could treat this scenario as an opportunity for additional code cloning and specialization without a penalty with respect to I-CACHE misses.

3.2 I-CACHE Benchmark

The main idea behind the I-CACHE benchmark is exactly the same as for the D-CACHE benchmark: the benchmark performs a fixed number of memory accesses across a varying memory footprint size and observes changes in running time. When

```
1 for i ← 1 to N by 1 do
2     instruction-i;
3 endfor
```

Figure 3.1: Original Loop

```
1 for i ← 1 to N by 2 do
2     instruction-i;
3     instruction-(i + 1);
4 endfor
```

Figure 3.2: Unrolled Loop

a level of cache becomes full, the cache misses for that level will increase, causing the average access latency to increase. The D-CACHE benchmark performs a constant number of explicit loads from a variable-sized data footprint. The I-CACHE benchmark, on the other hand, performs a constant number of instruction fetches (*i.e.*, an implicit load from instruction memory) from a variable-sized instruction footprint. In short, the benchmark holds the dynamic instruction count constant while varying the static instruction count. This approach reveals the instruction-cache hierarchy, since performance should degrade each time an instruction footprint fills a level of I-CACHE.

A simple loop-unrolling example illustrates the underlying mechanism of the I-CACHE benchmark. Consider the code in Figure 3.1—the loop on the left performs approximately N instruction fetches from an instruction footprint of size 1.[1] The unrolled loop, on the right in Figure 3.2, performs approximately the same number of instruction fetches (*i.e.*, N), but the size of the instruction footprint twice as large (*i.e.*, 2). In theory, we could continue unrolling the loop with increasingly larger unroll factors. We would expect to eventually find a loop body that exceeds the instruction cache, causing the instruction cache misses to increase and the performance of the loop to decrease. However, we only use the loop-unrolling approach for illustration, because in practice there are several obstacles to its success.

First, loop unrolling does not scale. As we increase the size of the loop body, the running time of the vendor compiler may not scale linearly; for large unroll factors

[1]For clarity, this example ignores loop overhead and boundary cases.

the compile times may become prohibitively expensive. Worse yet, the compiler may treat large loop bodies differently than small loop bodies, confounding the expected results. For example, the compiler may further unroll small loops but not large loops, effectively changing the experimental parameter that we're trying to control.

The second issue with the unrolling example is that the loop body expresses straight-line code, for which instruction prefetching is trivial. The only control flow is the backward branch at the end of the loop, which is easily predicted. If the underlying hardware employs even the most basic branch prediction and instruction prefetching, then much or all of the instruction fetch latency may be hidden. This outcome directly contradicts our goal of measuring the instruction fetch latency. Instead, we should employ techniques that impede branch prediction and instruction prefetching, allowing the I-CACHE benchmark to observe the entire fetch latency for each instruction.[2]

We address these challenges with modularization and indirection. The benchmark generates a set of functions, or *kernels*, of equal size. Each kernel is a single block of straight-line code. A main driver loop executes a set of kernels indirectly through a function pointer. Figure 3.3 shows the driver loop on the left and the set of kernels, or the *kernel working set*, on the right. The kernels are of equal size, so they each represent a fixed number of dynamic and static instructions. We hold the dynamic instruction count constant by fixing the number of iterations in the driver loop, which corresponds to the total number of kernel invocations. We vary the static instruction count by varying the kernel working set, or the set of distinct kernels that the driver invokes. This approach is analogous to the D-CACHE benchmark, in which the total number of data accesses is fixed while the size of the data footprint is varied. In both

[2]Although hardware prefetching can often hide instruction-fetch latency in actual programs, especially for straight-line code, it is still very important for the compiler to understand and effectively use the I-CACHE hierarchy. Prefetching does not reduce bandwidth to higher levels in the hierarchy, and it often increases it. Further, managing bandwidth is becoming especially important for current and future multi-core processors with shared resources, such as caches and busses.

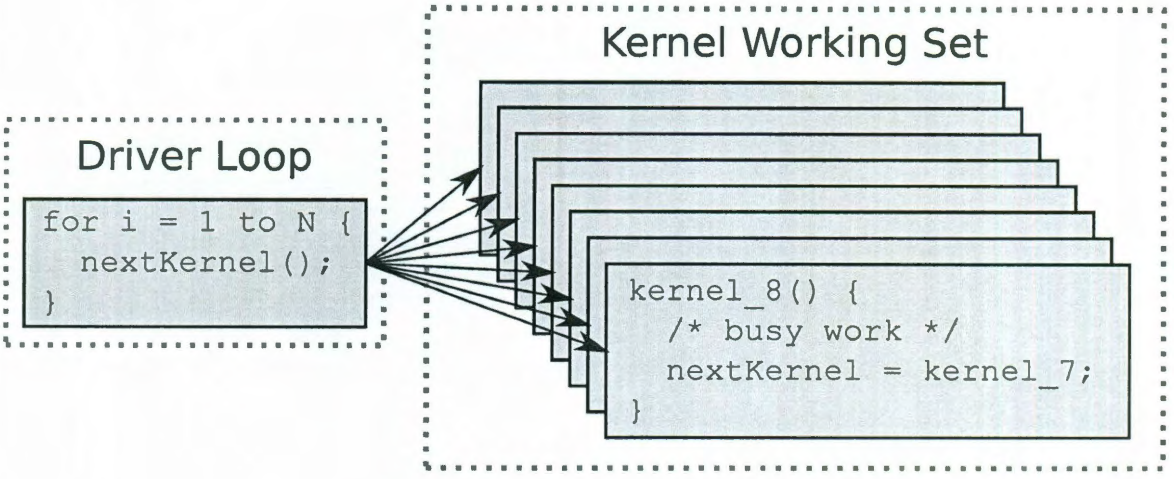


Figure 3.3: Kernel Working Set

benchmarks, the total number of memory accesses should be several times larger than the size of the memory footprint; this ratio allows the benchmark to reach a steady state that amortizes the cost of compulsory cache misses. The modularized approach addresses vendor-compiler scalability through separate compilation. We generate kernels in separate files, limiting the number of kernels in each file, ensuring a linear increase in compile time for additional kernels.[3] Also, the individual kernels are treated uniformly by the vendor compiler, since the code for all kernels is nearly identical. Finally, the indirect function call in the driver loop defeats the hardware features that hide instruction fetch latency. The branch predictor is unable to predict the target of the kernel invocation, since a single call-site targets a very large number of functions. The instruction prefetcher is unable to fetch instructions for the next kernel until the target is known; thus, prefetching is effectively limited to the size of the kernel.

The benchmark utilizes two global function pointers to control kernel traversal: `firstKernel` specifies the first kernel in the cycle and `nextKernel` specifies the next kernel in the traversal. `firstKernel` is initialized once before the driver loop begins

[3]Although the linker must process all object files at once, we have not experienced scalability issues with the linker. The narrow functionality of linkers should allow them to scale appropriately.

fetch operation, simply executing a region of instructions triggers an implicit fetch for each instruction.[4] Therefore, each kernel must ensure that its entire body of code is executed. Control flow should be avoided because, unless all paths are executed, branching might prevent some instructions in the kernel from being fetched. Predicated instructions may be useful if they are fetched but not executed, but there is no portable mechanism for specifying predicated instructions. Our benchmark generates simple, straight-line code for all kernels.

The next requirement for the kernel function body relates to a processor's instruction fetch and prefetch mechanisms. Processors are designed to fetch and issue sequential (*i.e.*, straight-line) instructions very efficiently—processors that support high levels of instruction level parallelism (ILP) usually fetch multiple instructions per cycle. In addition, since all instructions in a block of straight-line code are guaranteed to execute, prefetching ahead of the current program counter is trivial. These hardware optimizations are designed to keep the processor's computational units busy by hiding instruction-fetch latency. Unfortunately, the goal of the I-CACHE benchmark is to detect latency. If the processor is able to fetch the kernel body faster than it executes, then our benchmark will fail because it will detect uniform latency, regardless of the size of the instruction footprint. Instead, the kernel body must contain a sequence of instructions that executes faster than it can be fetched, allowing variations in fetch latency to be observed. For a RISC architecture, where all instructions are the same size, each kernel should maximize instructions per cycle (IPC)— this places the most stress on the instruction-fetch unit. But for a CISC architecture, which allows for variable-width instructions, the kernel must balance high IPC with the width of the instructions. That is, the kernel must maximize the *instruction memory throughput*, or the rate at which instruction memory is consumed by the processor. This argues for a kernel that contains a sequence of wide (*i.e.*, multi-byte) instructions that

[4]Technically, a branch can be thought of an an explicit instruction fetch, but for our purposes the implicit instruction fetch between sequential instructions is sufficient.

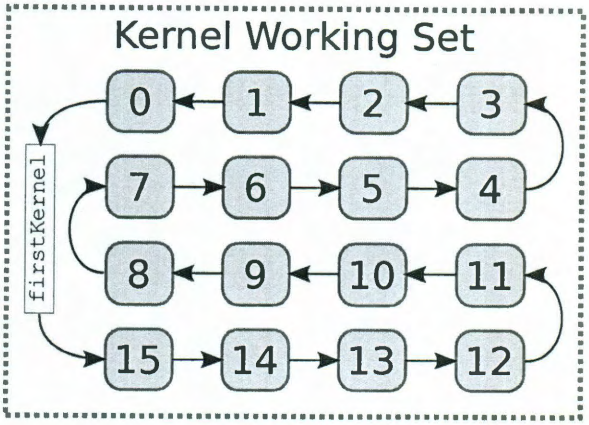


Figure 3.4: Cyclic Kernel Traversal

and remains constant throughout the test. Each iteration of the driver loop invokes the function to which `nextKernel` points; thus, `nextKernel` is updated once per iteration. The last instruction in each kernel updates `nextKernel` to point to the next kernel in the cycle. Figure 3.4 shows the traversal order through a working set of 16 kernels. $Kernel_0$ is always the last kernel in the cycle, and it restarts the traversal by assigning `firstKernel` to `nextKernel`. In the example the value of `firstKernel` is $kernel_{15}$. Every other $kernel_i$ assigns $kernel_{i-1}$ to `nextKernel`, which is shown in the example as directional edges between the adjacent kernels. The kernel traversal can be thought of as a repeated "count down" from `firstKernel` to $kernel_0$. This approach allows the size of the working set to be changed without recompiling the benchmark, simply by adjusting the value of `firstKernel`.

3.2.1 Kernel Contents

We previously indicated that the kernel function body should contain "busy work". Although there is definite flexibility in what each kernel may contain, this section will present the guidelines that all kernels should satisfy.

The goal of a kernel is to access a fixed-size region of memory through the processor's instruction-fetch mechanism. While we cannot explicitly issue an instruction

achieve a very high IPC.[5] Unfortunately, these two characteristics often oppose one another—fast instructions tend to be small while large instructions tend to be slow. Additionally, maximum IPC may sometimes only be achieved with a mix of several different types of instructions (since parallel functional units often support different instructions). Thus, the easiest way to find kernels with high instruction-memory throughput is with empirical search. Our benchmark generates a collection of kernel variants and evaluates them by invoking the I-CACHE benchmark with a single kernel, once for each variation. The algorithm selects the kernel variation that achieves the largest instruction-memory throughput, computed as bytes per second. The search space includes kernels of varying size with varying instruction-level parallelism for integer and floating point instructions.

Another trick for maximizing instruction memory throughput is to include numeric constants in the kernel computation, because such constants are often embedded directly in an immediate field of an instruction. An immediate value can be thought of as a data access that occurs through the instruction-fetch mechanism rather than the data-load mechanism. Thus, by including numeric constants in the instruction stream, we increase the pressure on the instruction fetch unit by forcing it to load both instructions and data through the instruction memory hierarchy. The size of these constants can also be adjusted to require different bit widths for the immediate field of an instruction. The benchmark includes in the search space kernels that contain different sized numeric constants.

3.2.2 Determining Kernel Size

Another challenge that the I-CACHE benchmark faces is determining code size. The ultimate goal of the benchmark is to determine the size of the levels of I-CACHE, in

[5]The processor's specific mechanism for achieving high IPC is of little importance. For example, the processor may employ pipeline parallelism, functional unit parallelism, or even just fast sequential execution.

bytes. But the benchmark, as we've described it so far, uses a more abstract unit of measure: the kernel. The benchmark can determine the number of kernels that fit into each level of cache, but we must be able to translate this abstract measurement into absolute size, or bytes. Thus, we must be able to determine the code size of a kernel.

In general, the C language provides no mechanism for determining code size. But given many similarities between implementations, especially for POSIX compatible systems, we have found several techniques that correctly estimate code size on many systems. Although none of these techniques is guaranteed to work on all systems, we have found that on all of our test systems at least one of the techniques always provides a suitable estimate of code size. The rest of this section describes our several techniques for determining code size.

POSIX `nm` Utility The POSIX `nm` utility displays symbol information for an object file or executable file. If the utility is available and standards compliant, we may be able to use it to query the symbol offsets in our I-CACHE benchmark. Given the address offset of each kernel, we can compute the size of a single kernel as the difference between its offset and the offset of the next kernel. This measurement should not underestimate the kernel size, but it may overestimate the size because the object file may include padding and other overhead between kernels. Unfortunately, there are several reasons why this approach may fail. First, if symbol information is stripped from the final executable, then kernel offsets will not be available. Second, if the compiler or linker employs name mangling on symbol names, then the symbol names may not be identifiable as kernel names. Finally, if the compiler or linker employs an indirection table for function placement, then the symbol offsets may refer to the indirection table rather than the actual function offset.

Function Pointers Similar to the `nm` approach, the function pointer approach relies on computing the address-space distance between two adjacent kernels. Although the C language does not officially support function pointer subtraction, we have found that most systems allow it and produce the expected results. This allows the benchmark to easily compute the size of a single kernel as the distance between two adjacent kernels. We cannot guarantee that the kernels will be linked in any particular order or that they will even be linked contiguously; thus, a more robust approach sorts the starting addresses for each kernel and then computes the distances between adjacent addresses. A histogram of the distances should exhibit a peak at the actual kernel size; a trimmed mean of the addresses could be used to discard outliers and correctly identify the actual kernel size.

One problem that we experienced with the function-pointer approach occurred on the Intel Itanium system using the GCC compiler. This particular system uses an indirection table for all symbols, similar to the program linkage table used in ELF dynamic libraries on Linux.. Each function pointer references an entry in an indirection table, which itself references the actual function. Thus, the pointer subtraction approach erroneously computes the size of an entry in the indirection table rather than the size of the kernel. Although we cannot solve this problem without platform-specific information, we can detect when it occurs by noticing that kernels of different lengths appear to require the same small number of bytes. In this case, we should rely on other approaches for estimating kernel size.

Inline Assembly Labels A possible workaround for problems introduced by the function pointer indirection table is to insert explicit symbol labels at the start and end of each kernel. Although C does not support global labels, we can insert inline assembly code that specifies "head" and "tail" labels at the function boundaries. Then the kernel size can be computed by subtracting the address of the tail label from the

address of the head label. This approach requires knowledge of the assembly syntax as well as the compiler's inline assembly syntax, although the label syntax appears to be quite common across various assembly languages. This approach also has the disadvantage that it may underestimate the kernel size, because it may ignore function prologue and epilogue code depending on how the compiler treats inline assembly. But we can be certain that this approach will not overestimate the kernel size; perhaps it could be used as a lower bound on the kernel size.

Executable File Size The most robust and portable approach for estimating code size is to measure the file size of the resulting executable. The I-CACHE benchmark executable will contain code, data, and object overhead for the kernel functions and other benchmark and library functions. Given a single executable, we have no way to determine what fraction of the file size represents kernel code and how much does not. But, given a set of executables, each identical except for the number of kernels, we can use least-squares regression to compute the size of an individual kernel. Figure 3.5 shows kernel-size-estimation plots for four different architectures and operating systems. The x-axis shows the number of kernels in an executable and the y-axis shows the size, in bytes, of the resulting executable object file. Each plot contains experimental data points for executables containing from one to 128 kernels and a linear least-squares fit line. The y-intercept of the best-fit line represents the size of the non-kernel code and object overhead, while the slope of the line represents the size of an individual kernel. The results for the Intel Core i7 Nehalem system are interesting because the experimental results produce a rough step-function; this behavior appears to be caused by padding in the resulting object file, and differs from the other systems because it is the only system running Mac OS X (the others run Linux and SunOS). Mac OS X uses a different object format than the other systems. Despite these step-like results, the linear regression still identifies the appropriate

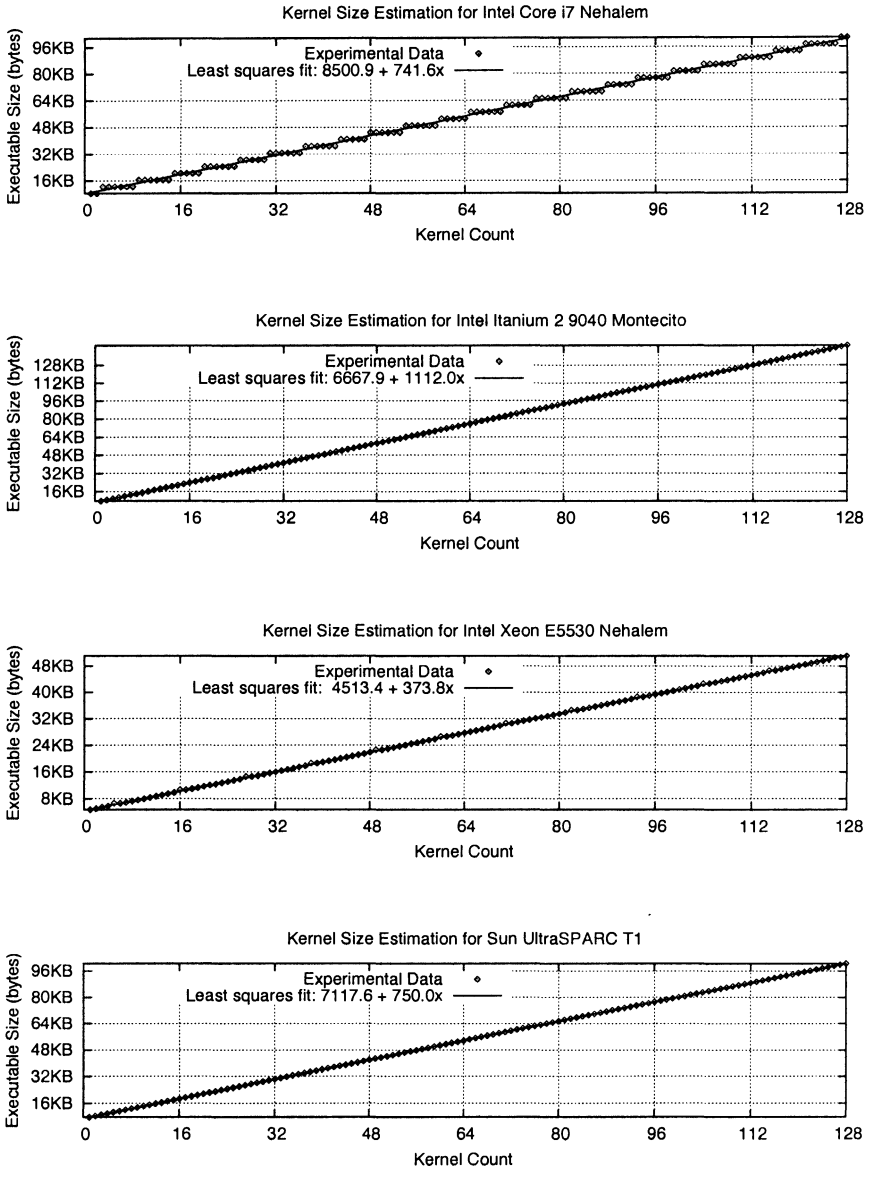


Figure 3.5: Estimating Kernel Size with Linear Regression

slope of the line.

This linear regression approach should never underestimate the size of a kernel, but it may overestimate the size if the object overhead for each kernel is significant.

Empty Kernel Comparison Finally, the last technique for estimating the kernel size is a derivation of the file size estimator. Instead of varying the number of kernels in the test executables, we can vary the length of the kernel. Specifically, we generate one executable with a large number of normal kernels, and another executable with the same number of empty kernels. The size difference between the two executables, divided by the number of kernels in each, should indicate the size difference between a regular kernel and an empty kernel. This technique may result in a slight underestimation of the actual kernel size, because an empty kernel still occupies some space for the function prologue and epilogue. Thus, the empty-kernel estimator should establish a lower bound on the actual kernel size.

These file-size techniques for estimating kernel size are particularly sensitive to object overhead. For example, debugging information may significantly increase the size of the executable, artificially increasing the kernel size estimate (since the debugging information does not represent executable code). There are several ways in which we can avoid this. First, we specify that the benchmark should not be compiled with symbolic debugging information enabled; for a POSIX-compatible c99 compiler, this means that the `-g` flag should not be used. Additionally, we can apply the POSIX `-s` flag to explicitly strip any additional non-essential symbolic or debugging information from the resulting executables. Finally, we can process the resulting executable with the POSIX `strip` utility, if available, to remove any symbolic or debugging information that remains after applying the previous safeguards.

Since the kernel-size estimator techniques described in this section are not guaranteed to work in all cases, and since many estimators inherently produce underes-

timates or overestimates of the actual kernel size, we have experimentally evaluated the effectiveness of each technique. Refer to Section 3.4.3 for the full results.

3.2.3 Kernel Traversal Order

The indirect function call in the driver loop prevents the hardware from prefetching code for the next kernel until the current kernel finishes. This is important to prevent the processor from hiding the instruction-fetch latency. However, if the kernels are traversed sequentially in order of object layout, then the access pattern will still exhibit strong spatial locality between kernels. (The sequential nature of individual kernels implies high spatial locality within a kernel.) If the processor prefetches beyond the end of one kernel, it will likely fetch instructions in the next kernel. Since such prefetching may hide the latency that the benchmark is trying to measure, we employ two mechanisms to prevent it: reversed traversal and localized randomization.

First, the kernels are generally traversed in reverse. That is, the cycle begins at some kernel i and traverses kernels in order down to kernel 0, which is always the last kernel in the cycle; the process then repeats. This decision prevents a prefetcher from benefiting from speculative prefetching beyond the end of the current kernel. The access pattern is not simply a reversed contiguous pattern, however; each kernel is traversed sequentially forward, while moving from one kernel to the next is in reverse.

Second, we relax the reversed traversal-order and apply localized randomization. That is, we group the kernels into blocks that are traversed in reverse, but the kernels within each block are traversed in random order. One reason we use blocks is because the kernel traversal order is statically specified at compile time and needs to allow for a variety of kernel sizes. If we uniformly randomized all of the kernels, then small working sets would not contain a contiguous set of kernels. Randomizing within blocks of kernels ensures contiguity modulo the block size. That is, the entire working set is guaranteed to be contiguous except for the last block in the working set, which

may not be contiguous. As additional kernels are added to the working set, however, they will fill in the non-contiguous "holes" in the last block of the working set; when the working set size becomes a multiple of the block size, then the working set is guaranteed to be contiguous. We have found that block sizes of 32 kernels work well in practice.

An interesting extension to the localized randomization is to increase the block size as the working-set size increases. The I-CACHE benchmark does not test every possible kernel working set size from 1 to the maximum number of kernels; rather, it tests exponentially increasing working set sizes. This makes sense because the sizes of the levels of memory hierarchy are exponentially increasing in nature. But, once we recognize that only specific working set sizes are tested, we can constrain the localized shuffling algorithm to select block sizes that ensure contiguity while maximizing randomization. For example, consider testing working sets of size of 1, 2, 4, and 8; we can shuffle the kernels in block sizes of 1, 1, 2, and 4. This ensures that each working set that we test will be contiguous, since each working set ends on a block boundary. But, as the distance between working set sizes increases, the shuffling algorithm is allowed more flexibility in shuffling. Note that this approach does not account for TLB locality, since the randomized block size may grow well beyond the page size or superpage size. It may be beneficial to limit the blocksize to prevent unexpected impact from poor TLB locality.

Although the indirect function call does impede the hardware prefetcher, it does not stop it entirely when a branch predictor successfully predicts the target of the branch. A small working set creates a situation in which the indirect function call only targets a small number of addresses, which may be small enough to fit in the branch predictor's history buffer; in this case, the branch predictor will be correct most of the time and prefetching beyond the current kernel will be possible. However, as soon as the number of kernels exceeds the size of the branch predictor's history, then

the branch predictor will rarely or never correctly predict the branch target. After this limit is passed, prefetching beyond the current kernel becomes nearly impossible. Because branch predictors are designed to specialize for the common case in which an indirect branch frequently targets one of a small number of addresses, it seems likely that the branch predictor's history buffer will become saturated after the kernel working-set grows larger than a small number of kernels. If the branch predictor's history buffer becomes saturated before the kernel working-set exceeds the L1 I-CACHE, then a processor's branch predictor and instruction prefetcher will provide minimal improvement and should not affect the results of the I-CACHE benchmark.

Finally, an additional benefit of the hard-coded traversal order is that it minimizes the size of the benchmark's data footprint. We could construct a linked-list traversal approach similar to that one we use for the D-CACHE benchmark; but, doing so would introduce a data footprint that grows with the size of the instruction footprint. Since the I-CACHE benchmark aims to measure I-CACHE only, we should minimize the D-CACHE impact. With only two global function pointers, our hard-coded traversal order only uses a small and constant-sized data footprint for all sizes of instruction footprints. An interesting future experiment would be to test the impact of using a data-array to construct a dynamic traversal order. Perhaps, if the L1 data cache is a large data-only cache, and the kernel size is large enough, then we could completely hold the traversal array in L1 cache without impacting the results. For example, a 32KB L1 data cache can hold 2048 16-byte linked list entries (*i.e.*, a node containing a next data pointer and a function pointer, both 8-bytes). If the kernel size is 1KB, then we could traverse an instruction footprint up to 2MB before exceeding the L1 data cache. Using an instruction footprint larger than 2MB would cause the data array to spill into L2 unified cache, possibly confounding the I-CACHE results. Increasing the kernel size may allow access to larger instruction footprints without exceeding the private L1 D-CACHE.

3.2.4 Linker Limits

The benchmark, as described, appears to conform strictly to the C language standard (except where explicitly noted). Unfortunately, the I-CACHE benchmark encounters practical constraints with the compiler and linker, related to the unusually large executable that it generates. Compilers usually generate relocatable object files, which include information about the symbols defined and referenced in that object file. The linker then combines a set of object files by *relocating* each one to a unique position in the final executable. The object files contain relocation information that allows the linker to change the object's offset by modifying a small number of relocation fields, or entries. The relocation entries are of a finite width, however, and on many systems do not support relocations beyond some fixed upper-bound address.. Thus, if the final executable contains too many functions, then the linker will be unable to fit all of them into the space that is addressable by the relocation entries. In this case, the linker fails and is unable to create the final executable—we have observed this behavior on several older POWERPC G3 and G4 desktop systems, as well as a modern ARM embedded processor. The rest of this section describes two possible workarounds for this problem: (1) using additional compiler and linker flags and (2) determining the maximum executable size.

First, many systems provide mechanisms for increasing the size of the object relocation entries. For example, the compiler may need to generate position-independent code and the linker may need to use different relocation mechanisms. Usually the compiler and linker will provide flags for enabling these features. Potentially, we can specify that these flags be used to build the I-CACHE benchmark. Unfortunately, such flags are platform dependent and contradict our portability goal. Although we can allow such flags as an optional improvement to the benchmark, we would like to avoid placing a strict requirement on them.

To preserve portability, we need a backup in the absence of compiler or linker

flags to extend the allowable size of a program's text section. If no flags exist, or they are unknown, then the next best option is to generate a kernel with the maximum number of kernels that the compiler and linker support. We can determine this number empirically—start with the desired upper bound on the number of kernels and decrease until the compilation and linking succeeds. Additionally, we can employ a binary search to improve efficiency of this approach. The resulting executable should be just within the bound for the largest executable supported by the underlying system. Although this size may be much smaller than our original upper bound for testing I-CACHE footprints, it prunes the benchmark's search space based on the real limits of the system. In other words, the footprints that we are prevented from measuring are ones that we don't care about, because they are larger than the system will actually allow.

3.2.5 Distinguishing Between Cache and TLB

The D-CACHE benchmark, as described in Chapter 2, uses separate benchmarks to distinguish between cache and TLB. The TLB-only test employs a memory pattern that accesses only a single word per page, emphasizing the TLB effects over the cache effects. The cache-only test tiles the access pattern for the page size, effectively amortizing away the single TLB miss per page. Although the I-CACHE benchmark could theoretically apply these same separation techniques for the instruction cache and TLB, it doing so is much more difficult because we do not have word-level access granularity for instruction memory. Our modular kernel approach further increases the instruction memory access granularity to hundreds or thousands of bytes. Because of this, a TLB-only I-CACHE test is not possible. However, the I-CACHE benchmark generally achieves results similar to the D-CACHE cache-only test, because the execution pattern achieves significant spatial locality within each page. That is, sequential execution within a kernel generally results in a maximum of one TLB miss per page—

adjacent instructions in the same page will hit in the TLB and amortize the cost of the first miss. If the kernel size is particularly small (compared to the page size) and the randomization blocking factor is particularly large, then it may be possible that the TLB misses are not entirely amortized. In this case, the performance curve may exhibit degradation at locations corresponding to the instruction TLBs. Since it is difficult to distinguish between cache and TLB effects in the same curve, we would like to avoid measuring the TLB effects. We can do this by limiting the randomization blocking factor and ensuring that the kernel size is large enough to amortize the TLB misses.

It may be sufficient to characterize only the instruction cache and to ignore the instruction TLB. For one, program text inherently exhibits spatial locality, because of the sequential nature of execution. Although some programming paradigms, such as functional programming or object-oriented programming, may reduce the spatial locality within a program, even those programs will retain some spatial locality at a low level. Linkers place all text sections adjacent in memory, and object layout optimizations are known to improve spatial locality [41]. Additionally, a program's total instruction footprint is generally much smaller than its data footprint, and over time program sizes are growing at a much smaller rate than data input sizes. This is because programs solve large problems through repetition (*i.e.*, iteration or recursion) of a relatively small number of instructions; a larger problem can be solved by supplying a larger input data set and running the same program longer. Since the total instruction footprint is relatively small and exhibits good spatial locality, it will occupy many fewer pages than the data footprint. Fewer pages means fewer page-translation entries in the TLB, suggesting that it is difficult for a program to exhibit poor behavior with respect to instruction TLB. The bottom line is that while data access patterns can easily exhibit worst-case TLB behavior (and they do in practice), it is much less clear that instruction-execution patterns can exhibit worst-case TLB

behavior. We do not know of any practical examples where this happens. On the contrary, the instruction cache does affect performance for real applications, especially for the lower levels of cache.

3.2.6 Automatic Analysis

Just like the D-CACHE benchmark, the I-CACHE benchmark generates a set of data points that map memory footprint size to access latency. Consequently, we can use the same automatic analysis for interpreting the empirical I-CACHE results. The automatic analysis is described in full in Chapter 2; it uses density estimation, Gaussian filtering, isotone regression, and step-function approximation.

Unlike the D-CACHE benchmark, the I-CACHE benchmark does not measure individual instruction-fetch latency. Instead, it measures kernel latency, which isn't particularly relevant to a compiler. Since we can estimate the size of a kernel, we can compute the latency per byte of instruction memory; this may provide a useful metric of instruction latency if we know the average size of an instruction. Also, we can always compute the relative instruction-fetch latency between the various levels of cache. The compiler may find this metric useful for estimating the profitability of various code transformations.

3.3 U-CACHE Benchmark

The D-CACHE and I-CACHE benchmarks are both designed to measure the impact of various sizes of one type of memory footprint—data or instruction—while minimizing the impact of the other type. It is important to focus on one type of memory footprint at a time to avoid measuring contention in caches that are unified. This narrow focus allows each benchmark to determine the largest footprint, either instruction or data, that fits into a particular level of cache, regardless of whether or not that level of cache

is unified. However, after we characterize the I-CACHE and D-CACHE separately, we'd like to determine which of those levels are unified and which are disjoint. We can do this with a U-CACHE test that deliberately tests memory footprints that would conflict in a unified cache but coexist without contention in disjoint caches.

The D-CACHE and I-CACHE benchmarks detect both disjoint and unified levels of cache in their respective cache hierarchies. However, we cannot determine which levels are unified from this information alone. For example, a 32KB L1 cache found in both the I-CACHE and D-CACHE hierarchies is not necessarily unified—it may be two disjoint caches, one in each hierarchy. The U-CACHE benchmark must test a combined memory footprint that incorporates both data and instruction accesses, in an effort to detect any contention between the two footprints. The U-CACHE benchmark creates a data footprint and instruction footprint of equal size. It tests each footprint separately to determine its independent running time. Then, it tests the two footprints in the same test by interleaving their accesses—this results in a total memory footprint of twice the size of each independent footprint. If the sum of the durations of the independent tests is equal to the duration of the combined footprint, then we conclude that there is no contention between the two footprints. If, on the other hand, the duration of the combined footprint is greater than the sum of its parts, then we conclude there is contention between the two footprints.

The key behind this benchmark is to test independent and combined footprints that straddle the boundary between known levels of cache in the D-CACHE or I-CACHE hierarchies. For example, consider a system for which both the D-CACHE and I-CACHE benchmarks detect a 32KB L1 cache. We'd like to determine if this cache is the same physical cache, or whether each hierarchy has its own private cache. We construct two data and instruction footprints such that each footprint is less than 32KB but the sum of the two footprints is greater than 32KB. Consequently, if the 32KB cache is in fact unified, then the two independent footprints will fit in the cache but the

```
double ucache_loop(int n,         /* Outer loop bound */
                   int m_inst,    /* Inner instuction-loop bound */
                   int m_data,    /* Inner data-loop bound */
                   int size_inst, /* Instruction footprint size */
                   int size_data  /* Data footprint size */)
{
  /* Initialize instruction and data footprints */
  /* Warm up the caches with the both footprints */

  /* Run the experiment */
  int i, j;
  double start = start_clock();
  for (i = n ; i > 0 ; i--) {
    for (j = m_inst ; j > 0 ; j--)
      next_kernel();       /* invoke next kernel */
    for (j = m_data ; j > 0 ; j--)
      next_data_access();  /* access next data element */
  }
  double stop = stop_clock();
  return stop - start;
}

double ucache_driver(int n,         /* Outer loop bound */
                     int m_inst,    /* Inner instuction-loop bound */
                     int m_data,    /* Inner data-loop bound */
                     int size_inst, /* Instruction footprint size */
                     int size_data  /* Data footprint size */)
{
  double time_inst, time_data, time_both;
  time_inst = ucache_loop(n, m_inst,      0, size_data, size_inst);
  time_data = ucache_loop(n,      0, m_data, size_data, size_inst);
  time_both = ucache_loop(n, m_inst, m_data, size_data, size_inst);
  return time_both / (time_inst + time_data);
}
```

Figure 3.6: Unified Cache Benchmark Driver Loop

combined footprint will experience serious contention. On the other hand, if the cache is disjoint, then each footprint will easily reside in its own private cache regardless of whether it is being accessed independently or jointly—the combined footprint will experience no contention.

We must be careful how we combine the two independent memory footprints. If we access the footprints in series—by first accessing the data footprint in its entirety, followed by accessing the instruction footprint in its entirety—then we may observe misleading results. The data footprint will incur compulsory misses on the first access

to each element, but will then achieve a steady state. The instruction footprint will do the same. This situation does not expose any contention, because each footprint achieves steady state after a single compulsory miss for each access. Instead, accesses to the two footprints must be interleaved so that the steady state of each is interrupted by the other. The function `ucache_loop` in Figure 3.6 illustrates the main loop for the U-CACHE benchmark. There are two inner loops, one that accesses the instruction footprint and another that accesses the data footprint. Each of these inner loops accesses a subset of its corresponding footprint. The inner loop bounds for each loop should be configured so that the loop does not achieve reuse within its footprint; rather, the reuse should be carried by the outer loop, otherwise the accesses to the two data footprints would not be sufficiently interleaved. When true contention between the footprints exists, it is emphasized as the two footprints thrash for space in the cache—neither is allowed to reach a steady state. When the two footprints reside in disjoint caches, on the other hand, the fine-grained interleaving will not introduce contention and should not affect performance.

The function `ucache_driver` in Figure 3.6 shows how to compute the ratio between the sum of the duration of the disjoint footprints and the duration of the combined footprint. The algorithm makes three distinct calls to the `ucache_loop` function: (1) the independent instruction-footprint test is performed by specifying an empty iteration count for the data-footprint inner loop; (2) the independent data-footprint test is performed by specifying an empty iteration count for the instruction-footprint inner loop; (3) the combined instruction and data footprint test is performed by specifying normal bounds for both inner loops.[6] The ratio is simply computed as the duration of the combined test divided by the sum of durations of the disjoint tests. The loop bounds should be automatically tuned to meet the following criteria:

[6]Care should be taken to ensure that the compiler does not inline and/or specialize the `ucache_loop` function for the two call sites with constant parameters. Placing the functions in separate files or calling the function through a volatile function pointer should sufficiently limit the compiler's ability to optimize.

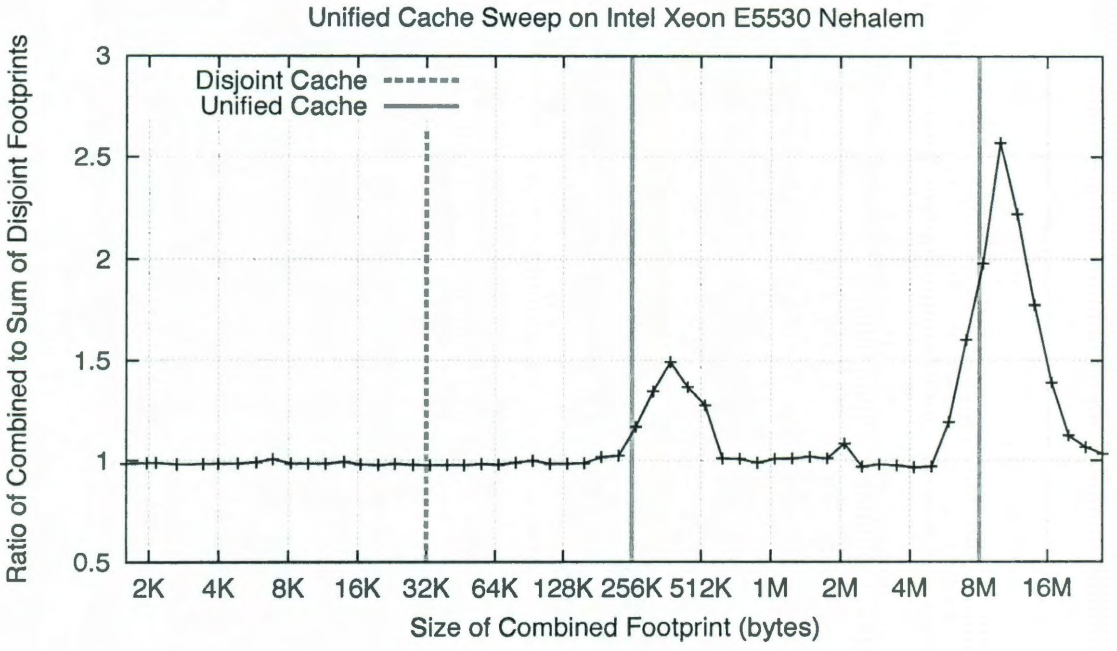


Figure 3.7: U-CACHE Sweep on Intel Xeon E5530 Nehalem

(1) the loop should execute long enough to meet the system's minimum timing granularity; (2) each footprint should have a reuse factor of at least 100; and (3) both disjoint footprints should execute for approximately the same amount of time.

As previously described, the U-CACHE benchmark can specifically test only the known levels of cache to determine if they are unified; alternatively, it can perform a sweep across memory sizes, similar to the sweep used by the I-CACHE and D-CACHE benchmarks. The specific tests result in more practical running times for the U-CACHE benchmark, but studying the sweep results is a useful exercise for improving our academic understanding. For the sweep we plot the ratio between the sum of durations of the independent footprints and the duration of the combined footprint. See the chart in Figure 3.7 for the results from a U-CACHE sweep on an Intel Xeon E5530 Nehalem processor. The x-axis represents the size of the combined footprint and the y-axis represents the ratio; a dashed vertical-line indicates a disjoint level of cache while a solid vertical-line indicates a unified level of cache. The leftmost region of the curve depicts a flat ratio of approximately 1.0, which indicates that

the combined footprint does not experience any contention relative to the sum of the disjoint footprints. This result is consistent with the actual hardware specification of a disjoint 32KB L1 cache. The ratio remains flat across the disjoint L1 cache, because the combined footprints do not experience any contention. The ratio remains flat even as the sweep progresses past the L1 cache, because both footprints are small enough to simultaneously reside in the unified L2 cache. However, the ratio begins to rise as the combined-footprint size approaches the unified L2 cache capacity. This indicates that the two independent footprints fit in the unified cache, but the combined footprint does not. Once the independent footprints no longer fit in cache, however, the ratio drops back towards 1.0—this indicates that both the disjoint and combined footprints exceed the L2 cache, but still fit within the L3 cache. As the combined size approaches the L3 capacity, however, we see another rise in the ratio, indicating that the L3 cache is also unified. Finally, as expected, the ratio eventually drops back toward 1.0 as the disjoint footprints no longer fit in L3.

The location and width of the peaks in Figure 3.7 correspond to properties of the cache. The peak begins to rise at the point when the combined footprint fills the effective capacity of the cache, similar to the D-CACHE or I-CACHE benchmarks. The width of the peak in this figure corresponds to the width of the transition between two levels of cache in the footprint-latency curve. If the transition between two levels of cache is very sharp, then the peak in the ratio plot will be very narrow. If the transition between two levels of cache is sloped and gradual, then the peak in the ratio plot will be very wide.

3.3.1 Automatic Analysis

The results of the U-CACHE benchmark require a different analysis than the one used for the D-CACHE and I-CACHE benchmarks. Fortunately, since the D-CACHE and I-CACHE benchmarks detect all levels in their respective hierarchy, the U-CACHE

benchmark only needs to determine which of those levels are unified. Further, since no known chip interleaves levels of unified and disjoint cache, the benchmark can stop testing after it identifies the lowest level of unified cache—it can assume that all higher levels of cache are also unified. This is entirely a performance optimization, however; it is trivial to allow the benchmark to continue testing all levels of cache.

The U-CACHE benchmark performs a direct test for each level identified by the D-CACHE benchmark, until detecting the first unified level. For each D-CACHE level, the U-CACHE benchmark returns a ratio of the duration of a combined instruction and data footprint to the sum of the durations of two disjoint instruction and data footprints. If the ratio is approximately one, then the level is disjoint; if the ratio is significantly greater than one, then the level is unified.

Unfortunately, interpretation of the ratio appears to be quite challenging, since classification requires comparison with a cutoff threshold. It is not clear that a single threshold will properly classify all cases—a threshold that is too high may introduce false negatives (*i.e.*, the analysis incorrectly concludes that a cache is not unified because the ratio was greater than one, but still below the threshold), and a threshold that is too low may introduce false positives (*i.e.*, the analysis incorrectly concludes that a cache is unified because noisy data causes the ratio to exceed the threshold). Consider the U-CACHE sweep in Figure 3.7: the ratio for the L2 unified cache peaks at approximately 1.5 while the ratio for the L3 unified cache peaks above 2.5. These U-CACHE ratios differ significantly because the ratio's magnitude depends upon the miss penalty for that particular level of cache. Since the miss penalty varies across systems and levels of cache, it can be expected that the cutoff threshold would also need to change.

Fortunately, we are able to estimate the cache's miss penalty for each level of cache by computing two more ratios. First, we compute the cost of accessing a double-sized data footprint over the cost of accessing the original data footprint twice. Then we

do the same with a double-sized instruction footprint. Since we know the capacity of this cache level, we know the double-sized footprint should exceed the capacity while the regular footprint should not—the ratio between the two estimates the miss-penalty for this particular level of cache. Now, we can compare the U-CACHE ratio to the miss-penalty ratio to determine whether the level of cache is unified. If the level of cache is unified, then the U-CACHE ratio should approximately match the miss-penalty ratio; if the cache is disjoint, then the U-CACHE ratio should be significantly less than the miss-penalty ratio.

Figure 3.8 shows the U-CACHE ratio along with the miss-penalty ratios computed by a double-sized data-footprint and a double-sized instruction-footprint. We can see that the miss-penalty ratios peak around each level of cache, but the U-CACHE ratio only peaks around the unified levels of cache. We can compute an average miss-penalty ratio by computing a weighted average of the D-CACHE and I-CACHE miss-penalty ratios—this is shown in Figure 3.9. This average miss-penalty ratio is meant to estimate the U-CACHE ratio that we'd expect to see if that particular level of cache was unified—each footprint in the combined footprint would experience a penalty approximately equal to the estimated miss-penalty. If the two ratios are approximately equal, then the cache is unified; if the U-CACHE ratio is significantly less than the average miss-penalty, then the cache is disjoint. Comparing the two curves in Figure 3.9 clearly indicates that the L1 cache is disjoint while the L2 and L3 caches are unified—this result matches the actual hardware specification.

With the estimated miss-penalty ratio, the automatic analysis is able to adjust the analysis cutoff-threshold based on the expected behavior for a unified cache.

Finally, recall from Section 3.2.4 that some systems may limit the number of kernels that the linker can place into a single executable, thereby limiting the size of the instruction footprint that the U-CACHE benchmark can create. Since the U-CACHE benchmark must create an instruction footprint of size equal to the level of

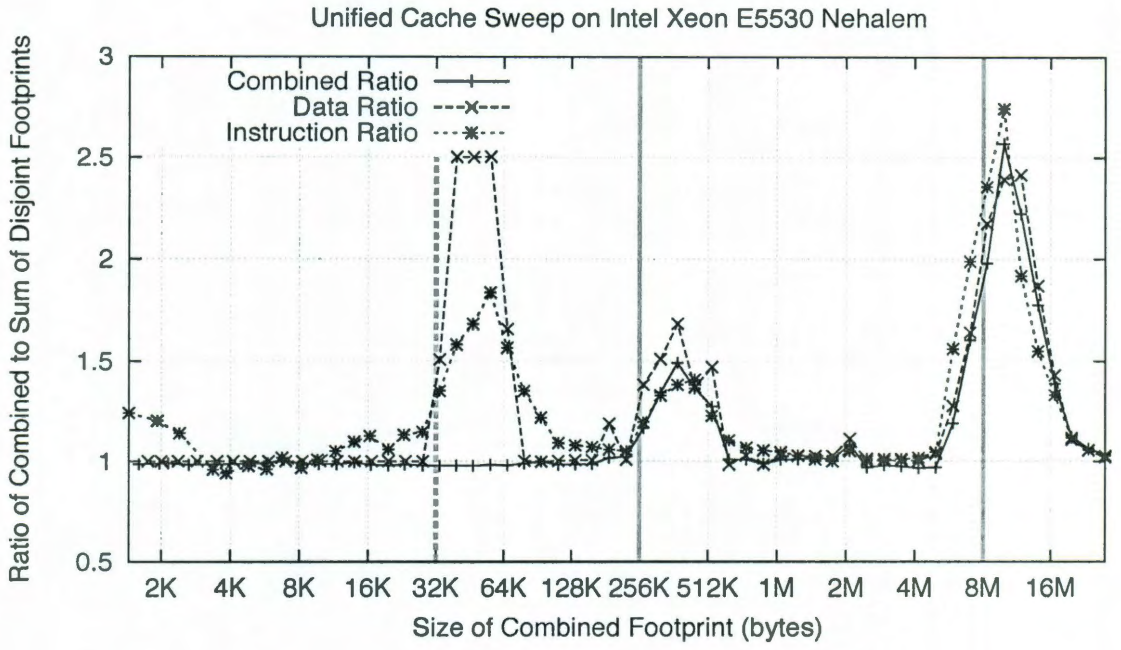


Figure 3.8: I-CACHE and D-CACHE Miss-penalty Ratios on Intel Xeon E5530 Nehalem

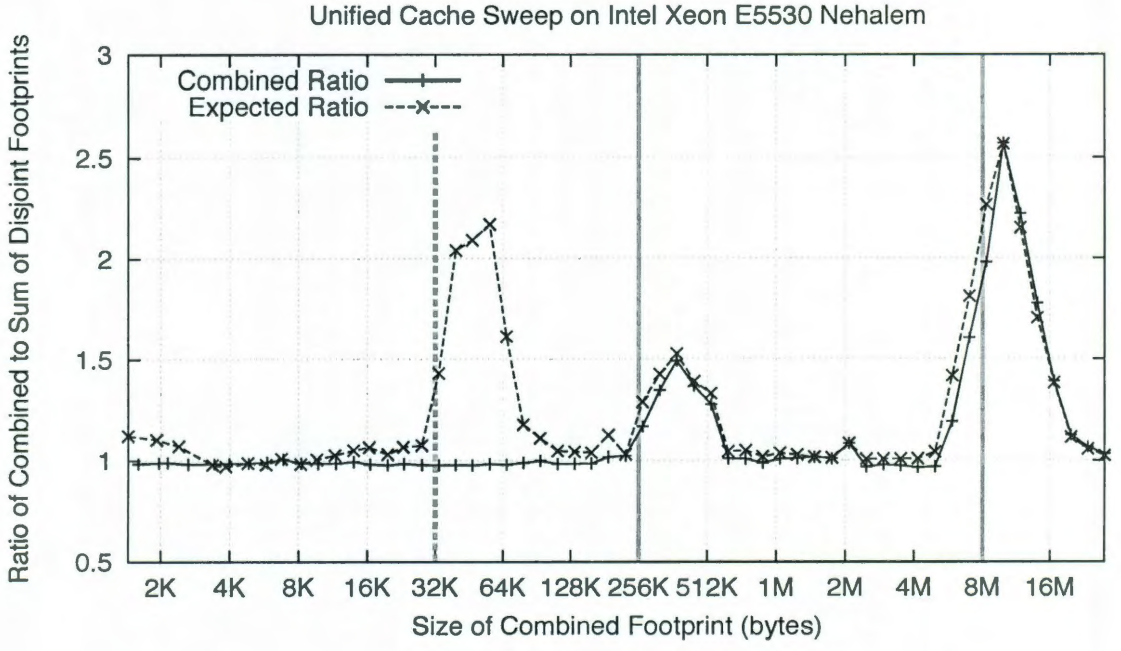


Figure 3.9: Average Miss-penalty Ratio on Intel Xeon E5530 Nehalem

Architecture
ARM926EJ-S
AMD Opteron 2360 SE Barcelona
AMD Opteron 275 Italy
AMD Opteron 6168 Magny-Cours
AMD Phenom X4 9750 Agena
IBM Cell (PS3)
IBM POWER7
Intel Core 2 Duo T5600 Merom
Intel Itanium 2 900 McKinley
Intel Itanium 2 9040 Montecito
Intel Itanium 2 9140N Montvale
Intel Pentium 4 Northwood C
Intel Xeon E5420 Harpertown
Intel Xeon E5440 Harpertown
Intel Xeon E5530 Nehalem
Intel Xeon E7330 Tigerton
Intel Xeon X3220 Kentsfield
Intel Xeon X5660 Westmere
PowerPC 7455 G4
PowerPC 750 G3
PowerPC 970FX G5
Sun UltraSPARC T1

Table 3.1: Testing Machines

cache that is begin tested, this linker limit could prevent the U-CACHE benchmark from testing large levels of cache. Fortunately, it seems unlikely that the linker limit would be less than the lowest level of unified cache—except for one system on which the L3 cache is the lowest level of unified cache, all of our testing systems have a unified L2 cache.

3.4 Results

This section presents the results from the I-CACHE benchmark on a variety of testing systems, which are listed in Table 3.1. The systems include many variants of Intel and AMD x86 processors; several POWERPC processors, including an IBM POWER7;

an ARM; and the IBM Cell processor in a Sony Playstation 3. All of these systems run some version of Unix.

In theory, we should be able to apply the automatic analysis from the D-CACHE benchmark to interpret the I-CACHE benchmark results. In practice, the I-CACHE results appear to be less clear than the D-CACHE results and the analysis is less often correct. The analysis may need slight modifications to properly handle the additional noise that is present in the I-CACHE results. Figure 3.10 presents results, in condensed form, for all of the test systems. The x-axis plots the size of the instruction footprint and the y-axis plots the time per kernel in nanoseconds (the axis labels have been omitted in the figure to save space). In general, the results for the rest of the systems resemble the expected shape; however, several anomalies become evident. A small spike on the leftmost region of the curve shows up on the two Intel Xeon Harpertown models, the E5420 and E5440; perhaps this results from the branch predictor or some sort of on-chip loop cache. This anomaly is easily filtered out during the smoothing stage of the analysis. On several other systems (AMD Opteron 6168 Magny-Cours, IBM POWER7, Intel Xeon X3220 Kentsfield, Intel Xeon X5660 Westmere, Intel Xeon E7330 Tigerton, and Sun UltraSPARC T1), we notice a slight increase in the curve that occurs much before the size of the physical L1 cache. Again, this anomaly may be related to the branch predictor. Unfortunately, the magnitude and shape of this type of anomaly makes it much more difficult to filter out during the analysis stage, likely increasing the number of false-positive conclusions.

Table 3.2 shows the results of applying the automatic analysis algorithm on the I-CACHE benchmark results. Figure 3.11 presents this same information in a graphical format. Each horizontal bar represents the physical I-CACHE hierarchy for a particular system, but the bar is normalized so that each level corresponds to the appropriate label on the x-axis. This normalization is linear between the major x-axis tic marks, but non-linear across the entire x-axis. The tic marks that appear superimposed on

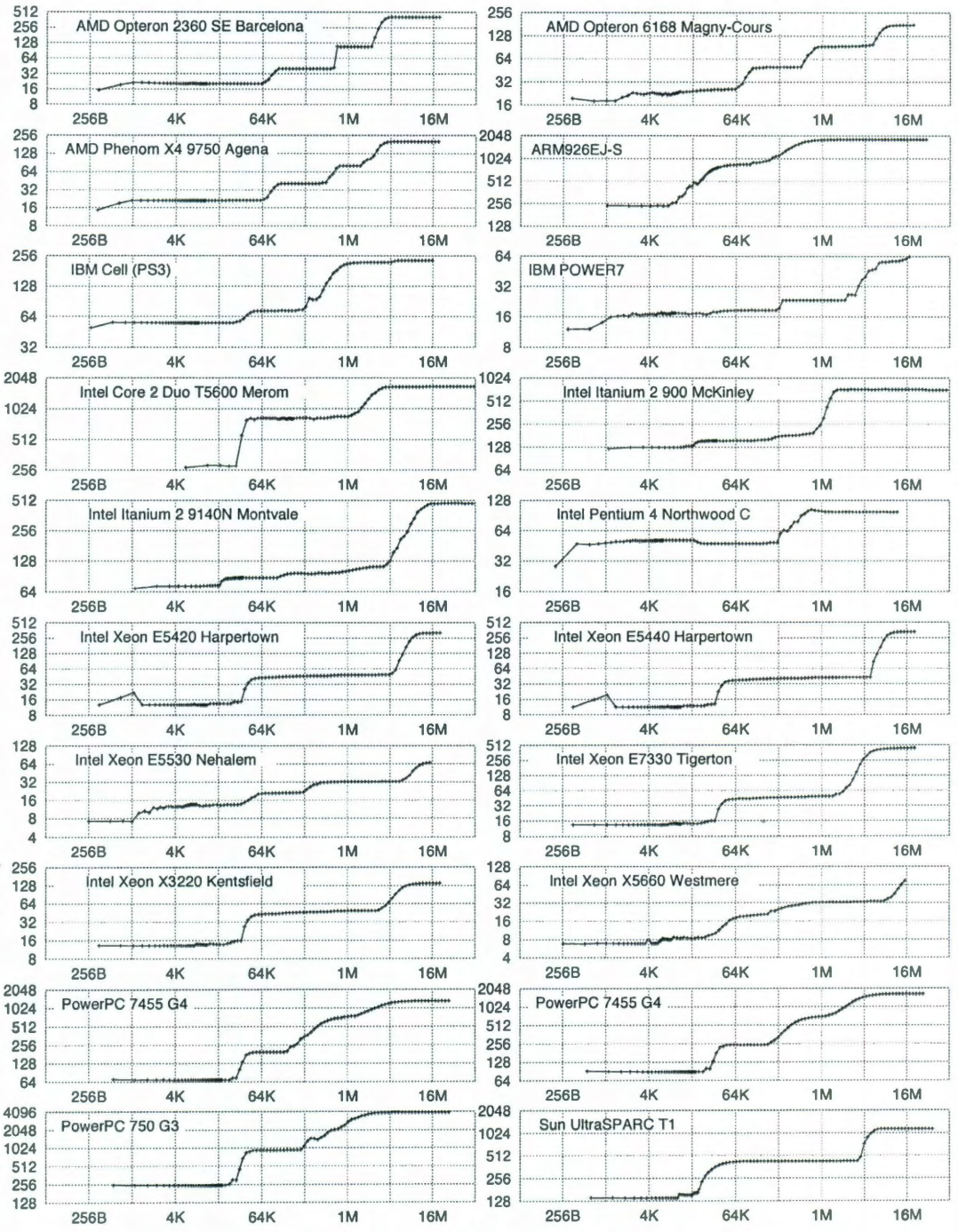


Figure 3.10: I-CACHE Results on All Systems

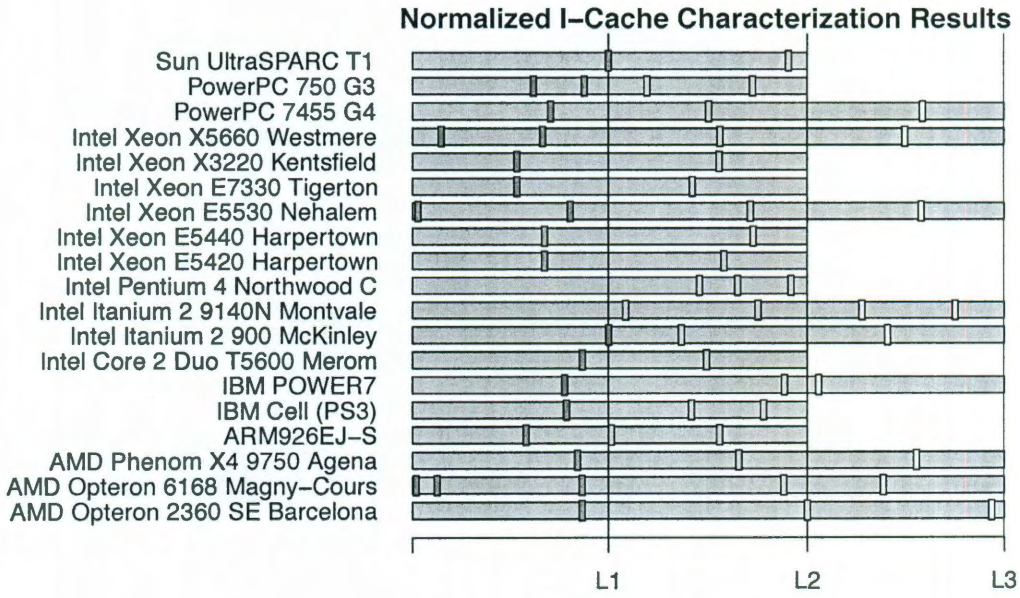


Figure 3.11: Normalized I-CACHE Results on All Systems

top of each bar correspond to the effective cache capacities returned by the I-CACHE benchmark. This plot allows easy evaluation of the effectiveness of the I-CACHE benchmark by observing where the effective cache-capacities fall in comparison to the physical cache-capacities. These results indicate that the I-CACHE benchmark is much less successful than the D-CACHE benchmark, described in Chapter 2.

In general, the main problem with the I-CACHE benchamrk is that the analysis identifies more levels of cache than actually exist. This is a consequence of the more noisy and jagged transitions in the I-CACHE results than in the D-CACHE results. More aggressive smoothing might solve the problem, but then we risk filtering out true features in the curve. On the Intel Pentium 4 the benchmark does not identify the L1 I-CACHE; but, the documentation for this processor describes the cache as a non-standard trace cache that holds up to 12KB of decoded micro-instructions [31]. This specialized cache apparently behaves differently than a normal cache because the results for this system, shown in Figure 3.10, are different from all other systems. On all other systems, the analysis correctly identifies the actual levels in the I-CACHE,

Machine	Source	L1	L2	L3	L4	L5
AMD Opteron 2360 SE Barcelona	Actual	64.0	512.0	2048.0		
	Measured	55.5	513.3	1948.4		
AMD Opteron 6168 Magny-Cours	Actual	64.0	512.0	10240.0		
	Measured	1.2	8.1	55.5	459.3	4241.6
AMD Phenom X4 9750 Agena	Actual	64.0	512.0	2048.0		
	Measured	54.0	358.3	1359.5	2120.7	
ARM926EJ-S	Actual	16.0	256.0			
	Measured	9.3	19.6	150.6		
IBM Cell (PS3)	Actual	32.0	512.0			
	Measured	25.2	233.5	406.9		
IBM POWER7	Actual	32.0	256.0	32768.0		
	Measured	24.9	230.3	2127.8		
Intel Core 2 Duo T5600 Merom	Actual	32.0	2048.0			
	Measured	27.8	1030.1			
Intel Itanium 2 900 McKinley	Actual	16.0	256.0	1536.0		
	Measured	16.5	104.6	775.5		
Intel Itanium 2 9140N Montvale	Actual	16.0	1024.0	9216.0		
	Measured	104.6	775.5	3288.2	7159.1	
Intel Pentium 4 Northwood C	Actual	12.0	512.0			
	Measured	242.0	337.7	471.5		
Intel Xeon E5420 Harpertown	Actual	32.0	6144.0			
	Measured	21.6	3595.3			
Intel Xeon E5440 Harpertown	Actual	32.0	6144.0			
	Measured	21.6	4490.3			
Intel Xeon E5530 Nehalem	Actual	32.0	256.0	8192.0		
	Measured	0.9	25.8	192.3	4827.4	
Intel Xeon E7330 Tigerton	Actual	32.0	3072.0			
	Measured	17.1	1322.2			
Intel Xeon X3220 Kentsfield	Actual	32.0	4096.0			
	Measured	17.1	2305.2			
Intel Xeon X5660 Westmere	Actual	32.0	256.0	12288.0		
	Measured	4.7	21.3	158.1	6190.5	
PowerPC 7455 G4	Actual	32.0	256.0	2048.0		
	Measured	22.6	145.4	1300.9		
PowerPC 750 G3	Actual	32.0	1024.0			
	Measured	19.8	28.1	225.8	752.9	
Sun UltraSPARC T1	Actual	16.0	3072.0			
	Measured	16.6	2784.4			

Table 3.2: I-CACHE Automatic Analysis Results

in addition to occasional false positives.

3.4.1 Native I-CACHE Benchmark

To better understand the challenging nuances of characterizing the I-CACHE, we performed experiments with a hand-tuned machine-code version of the I-CACHE benchmark. This benchmark dynamically generates a native machine-code instruction pattern in a variable-sized buffer. The flexibility of this approach allows us to create an instruction-memory access pattern that mimics the robust, cache-only pattern that we used the D-CACHE benchmark. The instruction pattern encodes a series of jump instructions that traverse the buffer in a randomized, sparse order. The sparseness is created by padding the jump instructions with non-executed regions of NOP instructions. This execution pattern reduces the spatial locality within a cache line by only executing one instruction per cache line. The randomized order is achieved by shuffling the targets of the jump instructions so that the traversal order is not sequential—just as in the cache-only pattern for the D-CACHE benchmark, we block the accesses for the page size to amortize TLB impact. This benchmark should reveal the best-case I-CACHE result that we can expect to achieve with any I-CACHE benchmark. Figure 3.12 shows the results for this native benchmark, and the actual I-CACHE levels are indicated by the vertical lines. For reference, Figure 3.13 shows the results obtained by the portable I-CACHE benchmark. The native I-CACHE benchmark does indeed provide much cleaner transitions, especially between the lower levels of cache, than the C version of the benchmark.

We'd like to understand why the C version of the I-CACHE benchmark falls short. There are two likely sources of the problem: (1) the kernel bodies do not achieve sufficient randomness and sparseness or (2) the indirect function-pointer framework obfuscates the results. We can test whether the indirect function-pointer framework is the problem by creating a hybrid kernel that mimics the sparse, random execution

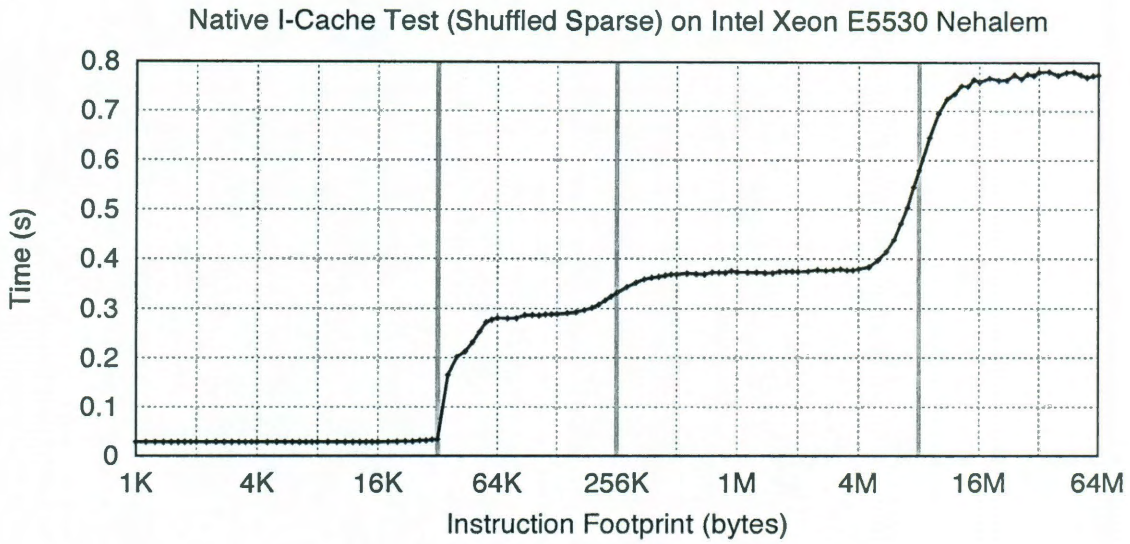


Figure 3.12: Native I-CACHE Benchmark Results

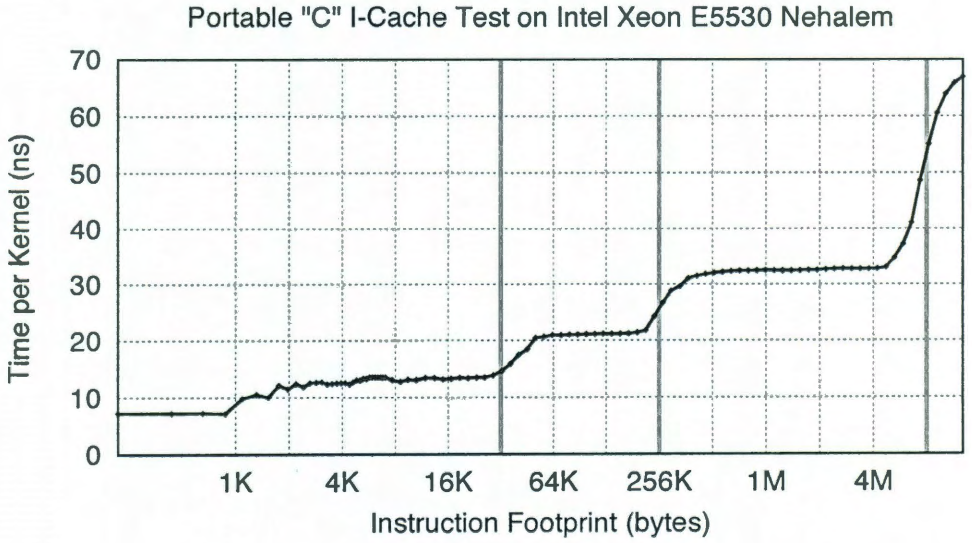


Figure 3.13: Portable I-CACHE Benchmark Results

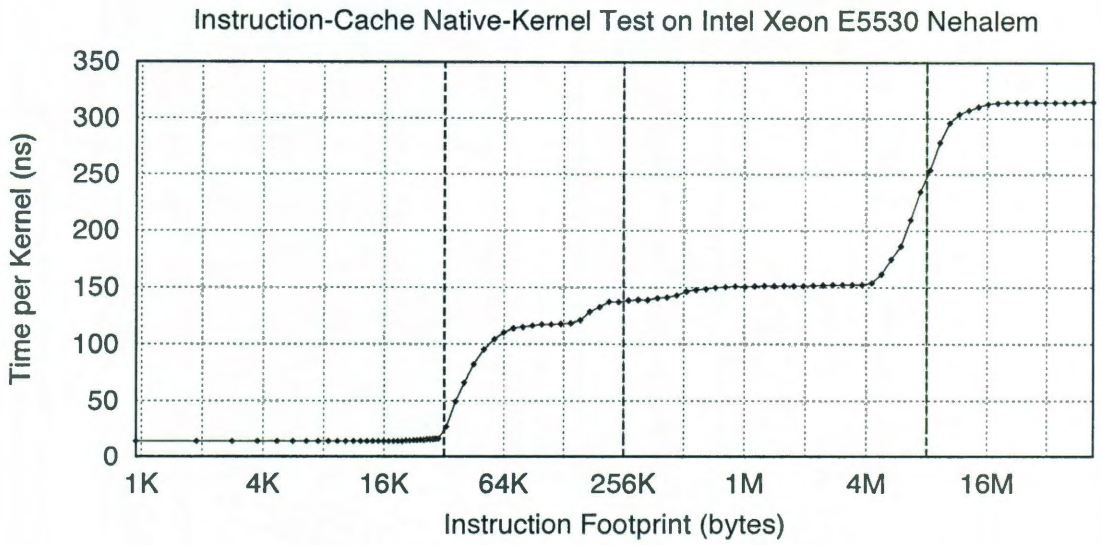


Figure 3.14: Native-Kernel I-CACHE Benchmark Results

pattern with inline assembly-code. Essentially, the kernel body is replaced with a sequence of inline assembly instructions that define the same pattern used by the native I-CACHE benchmark. Figure 3.14 shows results of this hybrid version of the benchmark. These results quite closely match the results of the native I-CACHE benchmark, shown in Figure 3.12—this suggests that the indirect function-pointer framework does not interfere with the operation of the I-CACHE benchmark. Thus, the main difficulty with writing a C version of the I-CACHE benchmark is that the C language cannot portably express a randomized, sparse execution pattern. [7]

3.4.2 U-CACHE Results

Table 3.3 displays the results from the selective U-CACHE benchmark, which performs a single U-CACHE test for each level identified by the D-CACHE test. The level of cache is indicated in the column marked "L". The size of the instruction and data

[7]Perhaps, using Duff's Device [30], it would be possible to construct a C function for which the compiler is likely to generate a random and sparse execution pattern. Unfortunately, this approach can never be guaranteed to work, since the compiler is always free to reorder basic blocks in the final executable.

footprints are under the "I" and "D" headers, which are both under the "Size" header. The running times for the data footprint, instruction footprint, unified footprint, double-sized data-footprint, and double-sized instruction-footprint are indicated in the middle group of columns, by the labels "D", "I", "U", "2D", and "2I". The ratios, which are described in Section 3.3.1, are shown in the rightmost group of columns labeled "Ratio". The "D" column shows the ratio of the double-sized data-footprint over two single data-footprints, and the "I" column shows the analogous ratio for the instruction footprint. These two columns estimate the relative miss-penalty for the data and instruction footprints at that particular level of cache. The "E" column, which displays a weighted average between the "D" and "I" columns, represents the *expected* miss-penalty for the combined footprint, if that level of cache is actually unified. Finally, the "U" column shows the ratio for the combined footprint over the sum of the data and instruction footprints. This value represents the actual penalty observed for accessing the combined, or unified footprint.

To determine if a particular level of cache is unified, we compare the unified ratio to the expected ratio. The expected ratio estimates the ratio that we'd expect to observe for the unified footprint if the level of cache is actually unified—otherwise, we would expect the unified ratio to be much closer to 1.0. Figure 3.15 presents the *contention factor* for each system and each level of cache. The contention factor is a measurement of how much of the estimated ratio is actually observed. The contention factor is computed as $\frac{(Ratio_U - 1.0)}{(Ratio_E - 1.0)}$, where $Ratio_U$ is the unified ratio and $Ratio_E$ is the estimated ratio. As the unified ratio approaches 1.0 the contention factor approaches to 0.0, indicating that the level of cache is disjoint. As the unified ratio approaches the expected ratio the contention factor approaches 1.0, indicating that the level of cache is unified. As we can see, the contention factor for the L1 cache on all systems is very close to 0—this result matches the actual machine specifications, because the L1 cache is always disjoint. In contrast, the L2 cache and beyond is unified on all of the

Machine	L	Running Time (s)					Ratio			
		D	I	U	2D	2I	D	I	E	U
AMD Opteron 2360 SE Barcelona	1	0.05	0.05	0.10	0.19	0.44	4.66	1.91	3.25	1.01
	2	0.32	0.40	1.32	1.43	1.49	2.32	1.77	2.02	1.83
	3	12.73	7.52	23.89	22.65	32.73	1.29	1.51	1.37	1.18
AMD Opteron 6168 Magny-Cours	1	0.05	0.05	0.10	0.20	0.48	5.09	1.98	3.49	1.01
	2	0.42	0.42	1.52	1.43	1.79	2.14	1.71	1.93	1.81
	3	7.97	7.91	48.24	51.91	41.72	2.62	3.28	2.95	3.04
AMD Phenom X4 9750 Agena	1	0.05	0.05	0.10	0.20	0.44	4.64	1.96	3.27	1.01
	2	0.29	0.29	0.91	1.01	1.29	2.24	1.74	1.99	1.57
	3	2.62	2.43	11.16	10.91	11.19	2.14	2.25	2.19	2.21
ARM926EJ-S	1	0.06	0.08	0.18	0.30	0.74	5.68	1.88	3.60	1.24
	2	2.08	2.11	6.42	5.42	7.26	1.75	1.29	1.52	1.53
IBM Cell (PS3)	1	0.05	0.05	0.10	0.13	0.45	4.97	1.29	3.06	1.02
	2	0.28	0.24	1.04	0.89	2.18	3.87	1.86	2.95	2.00
IBM POWER7	1	0.04	0.05	0.09	0.11	0.25	2.98	1.09	1.95	1.00
	2	0.05	0.05	0.18	0.13	0.42	4.28	1.27	2.76	1.77
	3	0.58	0.59	2.50	2.21	2.22	1.92	1.87	1.89	2.14
Intel Itanium 2 900 McKinley	1	0.05	0.05	0.10	0.12	0.28	3.00	1.19	2.06	1.01
	2	0.22	0.22	0.65	0.48	0.87	1.95	1.07	1.51	1.46
	3	0.93	0.93	5.85	4.88	6.20	3.31	2.62	2.97	3.14
Intel Itanium 2 9040 Montecito	1	0.05	0.05	0.10	0.12	0.28	3.00	1.16	2.05	1.02
	2	0.08	0.09	0.18	0.18	0.28	1.64	1.02	1.32	1.04
	3	3.71	3.03	27.48	29.84	17.89	2.41	4.92	3.54	4.08
Intel Itanium 2 9140N Montvale	1	0.05	0.05	0.10	0.12	0.28	2.99	1.18	2.06	1.02
	2	0.10	0.10	0.20	0.19	0.32	1.64	0.99	1.31	1.03
	3	4.21	3.04	15.89	10.64	19.98	2.37	1.75	2.11	2.19
Intel Pentium 4 Northwood C	1	0.04	0.05	0.11	0.10	0.81	9.25	1.00	4.86	1.12
	2	0.36	0.29	0.97	0.90	1.86	2.59	1.53	2.11	1.49
Intel Xeon E5420 Harpertown	1	0.05	0.05	0.10	0.21	0.50	4.98	2.12	3.55	0.99
	2	4.11	4.10	36.59	35.86	57.89	7.05	4.38	5.72	4.46
Intel Xeon E5440 Harpertown	1	0.05	0.05	0.10	0.21	0.50	4.99	2.11	3.55	0.99
	2	3.03	3.03	20.19	18.27	46.76	7.71	3.01	5.36	3.33
Intel Xeon E5530 Nehalem	1	0.05	0.05	0.10	0.20	0.25	2.50	1.98	2.24	0.98
	2	0.10	0.11	0.29	0.28	0.32	1.62	1.28	1.44	1.41
	3	3.74	3.43	8.70	8.88	11.70	1.57	1.29	1.43	1.21
Intel Xeon E7330 Tigerton	1	0.05	0.05	0.10	0.21	0.47	4.65	2.08	3.36	1.00
	2	1.39	1.47	5.76	5.75	6.74	2.42	1.96	2.18	2.01
Intel Xeon X3220 Kentsfield	1	0.05	0.05	0.10	0.21	0.47	4.65	2.05	3.35	1.01
	2	2.29	2.21	8.98	7.12	12.36	2.70	1.61	2.17	2.00
Intel Xeon X5660 Westmere	1	0.05	0.05	0.10	0.20	0.25	2.50	1.99	2.24	0.98
	2	0.08	0.08	0.23	0.22	0.26	1.70	1.29	1.49	1.41
	3	5.12	5.36	15.02	14.02	17.03	1.66	1.31	1.48	1.43
PowerPC 7455 G4	1	0.07	0.07	0.14	0.28	0.47	3.33	1.99	2.66	1.00
	2	0.78	0.80	3.46	3.35	3.51	2.26	2.09	2.17	2.19
	3	22.76	22.81	59.15	60.03	57.40	1.26	1.32	1.29	1.30
PowerPC 750 G3	1	0.21	0.29	0.51	1.30	4.44	10.41	2.25	5.71	1.02
	2	16.61	17.44	49.91	49.99	55.97	1.68	1.43	1.56	1.47

Table 3.3: Selective U-CACHE Results for All Systems

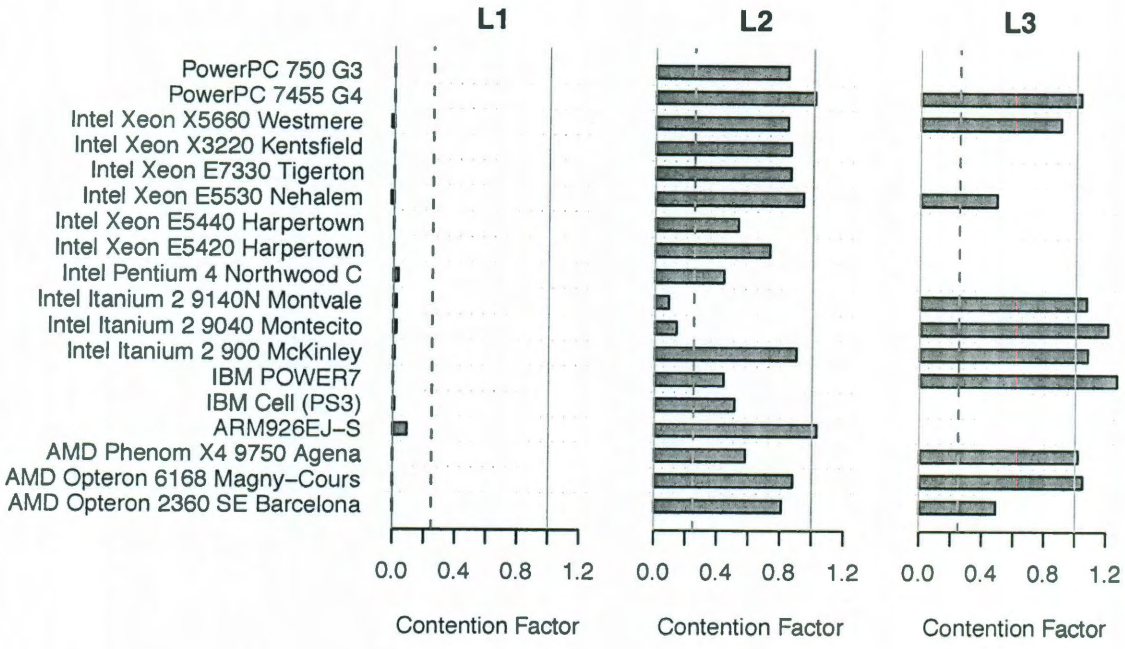


Figure 3.15: U-CACHE Contention Factor for All Systems

systems except for the Intel Itanium Montecito and Montvale systems. As expected, the contention factor is much closer to 1.0 for the unified levels of cache. The final step for the analysis is to classify whether or not a cache level is unified, based on the contention factor. Classification requires a cutoff threshold: levels with contention factors below the cutoff are classified as disjoint and levels with contention factors above the cutoff are classified as unified. The results are quite clear for disjoint levels of cache; so, a low cutoff of 0.25 should work well. The reasoning behind the cutoff is that disjoint levels of cache are easy to spot—if the cache isn't clearly disjoint, then it is likely unified. The threshold of 0.25 is shown as a vertical dashed line in the chart. From these results, a threshold of 0.25 appears to be a sufficient threshold for correct U-CACHE classification.

As Section 3.2.4 describes, platform-specific object formats may limit the number of kernels that the linker can place into a single executable, thereby limiting the size of the instruction footprint that the U-CACHE benchmark can create. This may

prevent the U-CACHE test from testing all levels of cache—we observe this behavior for the IBM POWER7, which has an effective L4 cache of approximately 20MB. Unfortunately, the linker is only able to successfully link 89,472 kernels, each approximately 300 bytes, for a total possible instruction footprint of approximately 25.6MB. Although the maximum footprint is large enough to create a single instruction footprint equal to the size of the effective L4 cache, creating a double-sized instruction footprint is not possible. Thus, the U-CACHE test cannot test the L4 cache on this system. Fortunately, the L2 cache and beyond are unified on this system, and the U-CACHE benchmark correctly identifies the L2 and L3 caches as unified, so the L4 cache is correctly assumed to be unified.

3.4.3 Kernel-Size Estimators

The various kernel-size estimator techniques described in Section 3.2.2 achieve different levels of accuracy. This section presents results that evaluate the effectiveness of the different kernel-size estimators. Several of the techniques produce different results depending on whether the source code is compiled with debug information and whether the final executable has been stripped of non-essential symbol information. We evaluate the kernel-size estimators in three different situations. The four estimators that we evaluate are "nm", "funcptr", "filesize", and "emptycmp"; these estimators are described in detail in Section 3.2.2. In each case, the actual kernel size is determined with the POSIX `nm` utility and verified by hand.

First, Figure 3.16 presents results for the ideal case in which the compiler does not generate any debug information and the final executable is stripped of non-essential symbol information. For this case, `nm` is unable to determine the kernel size because the symbol information has been stripped. The function-pointer estimator is the most effective for all cases in which it functions correctly; however, on the Intel Itanium 2 the function-pointer estimator significantly underestimates the kernel size. This

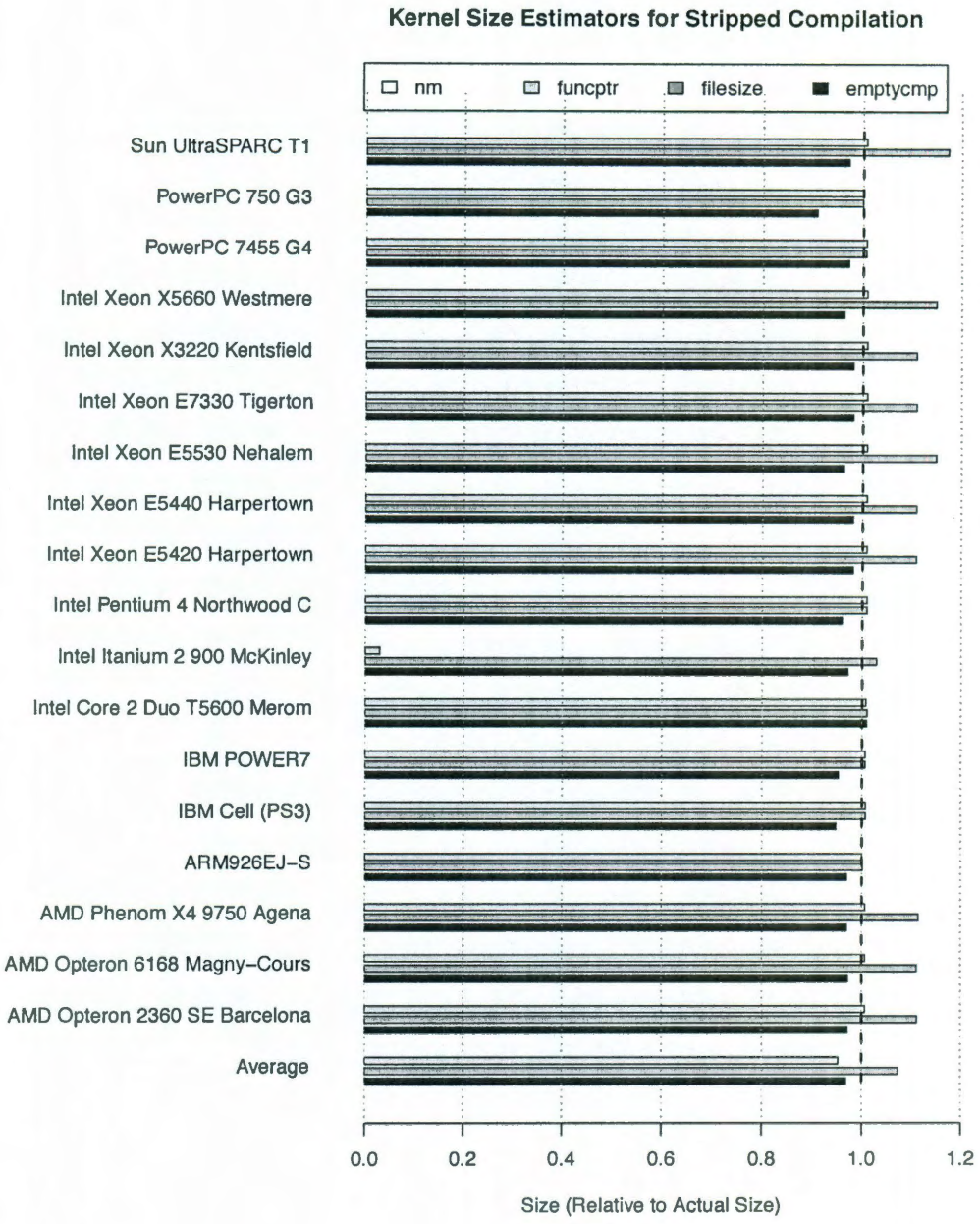


Figure 3.16: Estimating Kernel Size with Stripped Executables

underestimate occurs because the compiler on this system uses an indirection table for all function addresses; thus, the estimator returns the size of an entry in the table rather than the size of the kernel. The file-size estimator is the most consistent since it always produces a reasonable overestimate of the kernel size. On average, the file-size estimator over estimates the kernel size by just under 10%—this can be used as an upper bound on the kernel size. Finally, the "emptycmp" estimator slightly underestimates the kernel size in all cases—this can be used as a lower bound on the actual kernel size.

Figure 3.17 shows the kernel-size estimator results for the case in which the compiler performs standard optimization, without debug information, but the final executable is not stripped. In this configuration, the `nm` utility is able to perfectly determine the kernel size on all systems with a POSIX-compatible version of `nm`. This estimator is still unreliable, however, because not all systems provide a compatible version of `nm`. The function-pointer and "emptycmp" estimators perform the same as the last case in which the executable was stripped. In contrast, the file-size estimator produces even larger overestimates than when the executable is stripped, which makes sense because the extra object overhead results in a larger kernel-size estimate. Clearly, the file-size estimator is affected by the executable's object overhead, and stripping the executable is important for achieving a good estimate.

Finally, to complete the kernel-size estimator evaluation, Figure 3.18 shows the results for the case in which the compiler retains debug information, the compiler performs no optimization, and the final executable is not stripped. The `nm` and function-pointer estimators perform the same as in Figure 3.17. However, the file size and "emptycmp" estimators now grossly overestimate the kernel size, by up to a factor of 15! This result strongly indicates that these two kernel-size estimators perform quite poorly when the source code is compiled with debug information.

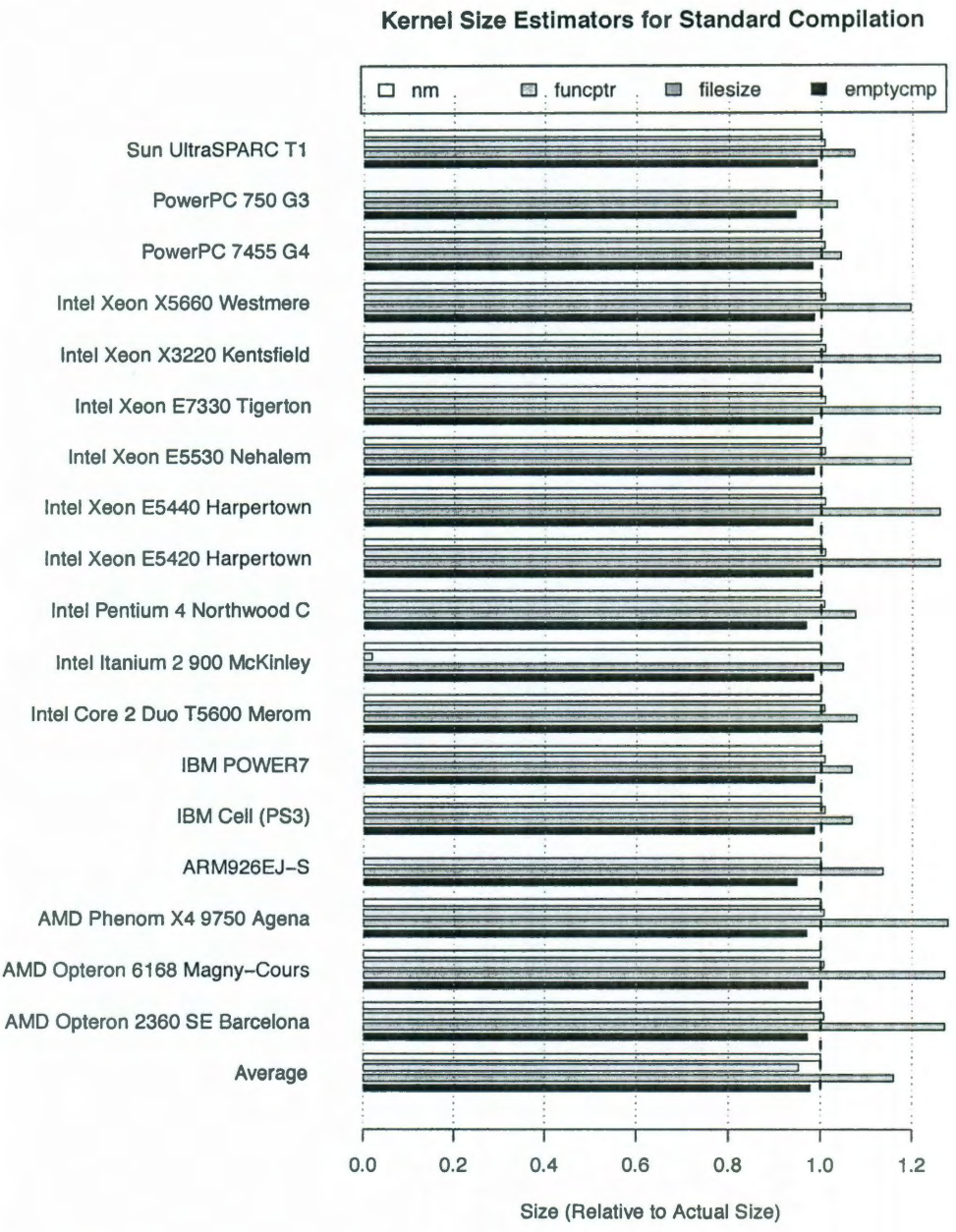


Figure 3.17: Estimating Kernel Size with Standard Executables

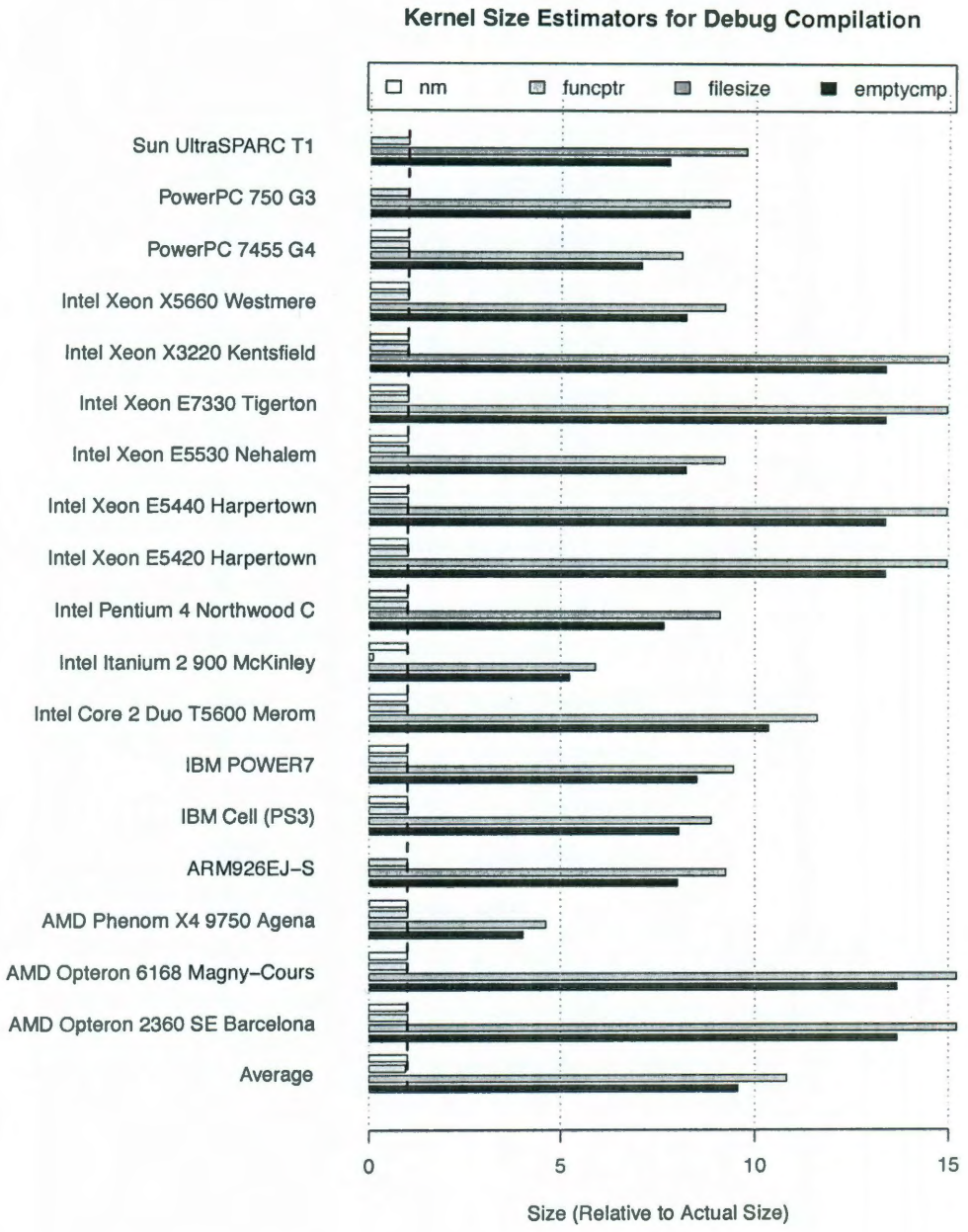


Figure 3.18: Estimating Kernel Size with Debug Executables

3.5 Conclusion

This chapter presented an I-CACHE benchmark to characterize a processor's instruction cache. Many of the concepts are extensions of the D-CACHE benchmark, but the I-CACHE context introduced several unique problems. Instruction memory cannot be explicitly accessed; thus, the benchmark implicitly accesses varying sizes of instruction footprints by varying the number of routines used in the computation. The approach is modularized to allow uniformity and scalability with the native compiler. Because we cannot portably query the absolute size of the instruction footprint, we implemented several approaches for estimating the code size. Finally, we introduced the U-CACHE benchmark, which combines data and instruction footprints to identify levels of cache that are unified.

Although it is possible to write the I-CACHE benchmark entirely in portable C code, we have shown that it is quite difficult to achieve consistent results across a variety of systems. We performed a thorough evaluation of a hand-tuned machine-code version of the I-CACHE benchmark, which provides insight into the best-case result that we can expect to achieve with any I-CACHE benchmark. The machine-code benchmark appears to indicate that sparse, randomized execution patterns are necessary to achieve consistent results. Unfortunately, these features are impossible to reliably achieve with portable C code.

Despite the deficiencies with the portable I-CACHE benchmark, the U-CACHE benchmark still operates as expected. The U-CACHE benchmark does not require a perfect I-CACHE access pattern to detect contention in a unified cache. It appears that any I-CACHE access pattern, regardless of how efficiently it accesses the I-CACHE, is enough to expose contention. This behavior makes sense, because the U-CACHE test doesn't need to detect contention in both footprints. Even if the instruction access pattern isn't sophisticated enough to clearly expose a level in the I-CACHE hierarchy, accessing that footprint still populates the cache—if the cache is unified, then

elements of the data footprint will be evicted. Since the D-CACHE access pattern is sophisticated enough to detect such contention, the U-CACHE test correctly concludes that the cache is unified.

Chapter 4

Intermediate Representation Annotation

4.1 Introduction

Automatic resource characterization, as described in Chapters 2 and 3, will enable compilers to adapt their optimization strategies quickly to changes in computer hardware. The resource characterization benchmarks are intended to be invoked once when the compiler is installed on a new system. The benchmarks fully characterize the system, providing a database of information that allows the compiler to tailor each program to the effective strengths of the underlying system. Undoubtedly, improving static compilation through resource characterization should be considered a vital technique for building an adaptable compiler. However, this approach assumes that the input programs will be amenable to static optimization. As HPC software grows in size and complexity, application programmers are beginning to employ software engineering techniques and paradigms that improve productivity while often impeding static compiler optimization. These constructs limit the compiler's ability to statically analyze and optimize the source code—in other words, static compilers

are becoming less effective for HPC software. This trend suggests that dynamic compilers may be better suited to program optimization than static compilers, because they can observe the dynamic context and behavior of the program before making optimization decisions. However, this chapter does not argue that static compilers should be abandoned in favor of dynamic compilers; rather, it recognizes that static compilers should remain the primary source of program optimization, but argues dynamic compilers should become the secondary source of program optimization. This chapter introduces *intermediate-representation (IR) annotation*, a low-overhead approach for dynamic compilation that allows the compiler to *selectively* optimize only the regions of a program for which static compilation failed to produce efficient code. IR annotation trades code growth for performance and flexibility: it maintains multiple program-representations and leverages the benefits of each. IR annotation is a static compilation technique that generates a fully-optimized native binary tagged with a higher-level compiler intermediate-representation of itself. The optimized binary achieves the speed of native execution immediately upon invocation, but the IR annotation allows aggressive runtime optimization that can leverage an existing compiler infrastructure. This technique is more portable than binary optimization systems and supports optimization across multiple programming languages. Just as resource characterization improves a compiler's adaptability with respect to hardware, dynamic compilation improves its adaptability with respect to software.

4.1.1 Motivating Case Study

The high-performance computing (HPC) community has historically focused on developing software that runs very fast and efficiently. This approach sometimes even precluded the use of software-engineering techniques and programming-language features that might incur a performance overhead. The HPC community has managed to pursue this strategy for several decades, but the increasing complexity of HPC

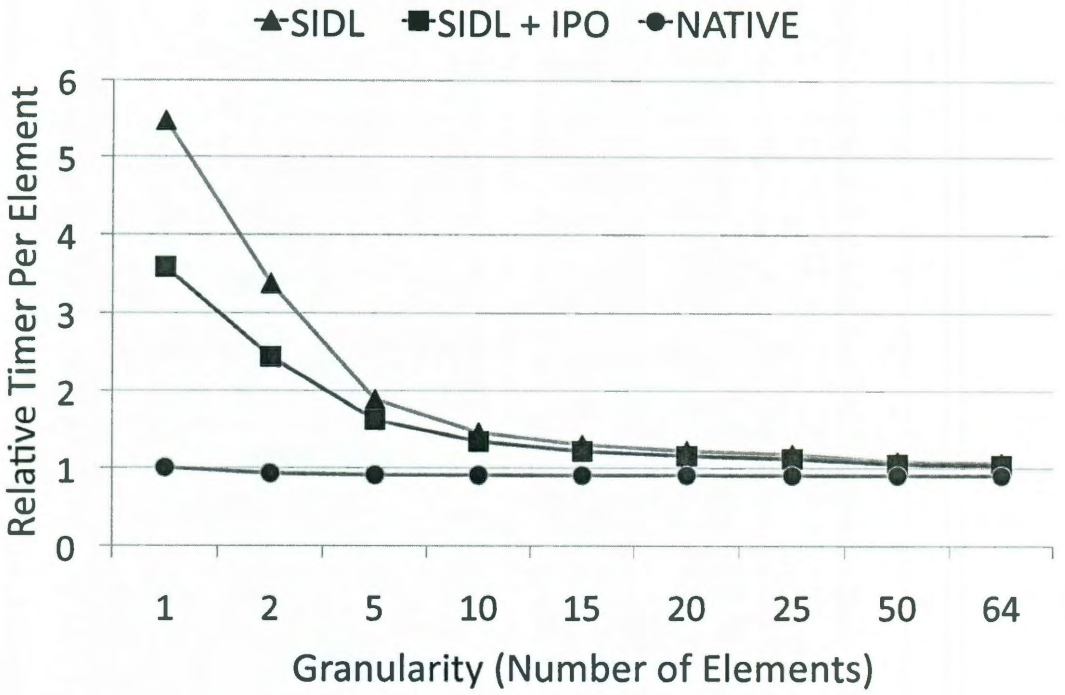


Figure 4.1: Mesh benchmark performance: SIDL vs. native

software and hardware is now forcing HPC developers to reconsider their development practices. Component-based frameworks for high-performance computing, such as the Common Component Architecture (CCA), are being developed in response to this growth in software complexity [7]. These frameworks promote the development of clear programming interfaces that allow easy code reuse, distributed development, and inter-language operability. The CCA-Forum also encourages the development and standardization of domain-specific programming interfaces, allowing application developers to select from a variety of different component implementations that support a given interface [1].

While the productivity benefits offered by component-based frameworks are desirable, using them sometimes incurs a performance overhead. Often the actual overhead is determined by the granularity at which a component is used. Consider the performance comparison for a mesh interface in Figure 4.1. The mesh interface, which is used for numerical calculations, allows mesh vertices and edges to be queried in bulk.

The Native mesh interface uses language-specific C pointers to query the mesh library in its native format; consequently, the overhead is small and the granularity has little impact on its performance. The component-based mesh interface, on the other hand, includes a language-independent interface, specified in *scientific interface definition language (SIDL)*, that uses a generic array API instead of C pointers [14, 17, 32]. SIDL allows scientific components to be defined with a language-neutral interface, while the implementation and use of such a component can employ one of several supported languages. Without careful programming, using the SIDL array API can result in an increase in memory allocation/deallocation and unnecessary copying. Figure 4.1 shows that the cost of querying a single mesh element through the SIDL interface is significant. If the element accesses are grouped, however, then the relative cost of querying through the SIDL interface decreases. This observation applies to component-based programming in general: a fine-grained use of components is more likely to incur unacceptable overhead simply because the computation does not overshadow the framework overhead. This work focuses on reducing the overhead for fine-grained component-based applications, allowing programmers to choose the most natural level of granularity without a concern for poor performance.

Historically, improving compiler optimizations has been an effective approach for reducing the overhead of programming-language abstractions [46]. Unfortunately, two features of component-based programming make it difficult for traditional static compilation to generate efficient code for these programs; those features are abstract interfaces to components and dynamic linking and loading of components. Abstract interfaces decouple a component implementation from its uses by only revealing a contract of what the component will do, but not how. One or more different components implementing the same abstract interface can be used interchangeably. This flexibility comes at a cost, however: procedure calls across components translate into indirect function calls that can significantly hinder inter-procedural analysis and op-

timization. If multiple implementations are available for a given abstract interface, the actual choice between these component implementations may not be decided until runtime. The use of dynamic linking and loading adds additional complexity because it allows new components to be provided at runtime. A static compiler is unable to optimize across components that are dynamically linked or loaded, simply because the code is unavailable at compile time. For these reasons, this chapter proposes using dynamic compilation to reduce the overhead for component-based applications. Abstract interfaces resolve to a specific component implementation at runtime, allowing a dynamic compiler to eliminate the indirect function call and to optimize across it. Because the entire program is present while it is executing, dynamic compilation is even able to optimize across boundaries of dynamically loaded components. Lastly, past experience shows that dynamic compilation can successfully improve runtime performance for object-oriented languages, a domain where abstract interfaces and dynamic linking are typical [19, 13, 5, 12, 39, 47].

To illustrate the potential benefits of inter-procedural optimization (IPO) across interfaces, consider the second line in Figure 4.1, labeled "SIDL + IPO". This experiment shows that overhead for using the fine-grained SIDL interface can be reduced from 5.5x to 3.5x simply by applying existing optimization techniques across the interface. We used LLVM's link-time optimization (LTO) [33], a technique that aggregates all compilation units into a single file before applying whole-program optimization. We were able to achieve this result statically because the mesh benchmark is not a full CCA application – that is, the interface was not abstract and the implementation was not a dynamically loaded component. For full CCA applications, dynamic optimization will be necessary to apply similar optimizations.

We are proposing a technique called *intermediate-representation (IR) annotation* that will allow programs written in several statically compiled languages to be dynamically optimized at runtime. IR annotation is a static compiler transformation

in which a high-level intermediate representation (IR) of a program is embedded inside an optimized native executable. This approach leverages the benefits of multiple program representations at the cost of increased program size. The resulting native binary functions as a standard executable and can execute at full native speed immediately upon invocation; yet, the high-level IR is available for a runtime system to perform dynamic re-compilation. The main disadvantage of IR annotation is that the resulting binaries will be larger.

4.1.2 Advantages of Dynamic Compilation

In addition to the benefits already mentioned, we believe there are several other reasons why dynamic compilation is complementary to resource characterization and static compilation. First, modern microprocessors are becoming increasingly difficult to model and predict. Long pipelines, branch predictors, hardware prefetching, and complex cache hierarchies are all features that make static compilation difficult. Even with an accurate characterization of the individual architectural resources, most optimization decisions involve balancing the trade-offs between several competing goals simultaneously. This complex and multi-dimensional profitability calculation can be very difficult to model accurately, leading to sub-optimal optimization decisions. Dynamic compilation can remedy this problem by continually monitoring a program's runtime performance and correcting the cases in which the static compiler made a poor decision.

With the advent of multi-core chips we see a new opportunity for performing dynamic compilation in parallel with a running program. Traditionally, the benefit of dynamic compilation is determined by subtracting the cost of the compilation from the speedup achieved by it. With multi-core chips, however, the compilation cost can be removed from the critical path of the program. With the predicted increase of cores to hundreds or even thousands, we believe that reserving a small number of

cores for dynamic compilation is entirely reasonable.

Finally, we also believe that recent improvements in light-weight performance profiling can further reduce the cost of dynamic compilation. As the profiling overhead approaches zero, dynamic compilation becomes a win-win technology: the system incurs insignificant overhead unless an optimization opportunity is discovered, in which case invoking the dynamic compiler is very likely to improve performance. That is, the framework should never degrade a program's performance, but in certain cases it may improve it.

Following the overall theme of this dissertation—*foundations for adaptable compilation*—the rest of this chapter will present the design and analysis of our IR annotation framework, which acts as a foundation for dynamic compilation. Section 4.2 describes the details of the technique and Section 4.3 presents experimental results measuring code growth. Although a full implementation and evaluation of a dynamic compiler is beyond the scope of this work, Section 4.4 describes the intended use of this framework. We propose a runtime system that can use the annotated IR to re-compile a running program. Lightweight performance monitoring will identify optimization opportunities [2], and the system will only invoke the dynamic compiler when a performance problem is exposed. Finally, Section 4.5 compares IR annotation to other related approaches and Section 4.6 completes the chapter with some final conclusions.

4.2 IR Annotation

IR annotation is a technique that we expect can be implemented in any static compiler. With IR annotation the static compiler will produce a native object file as expected, but the object file will contain a high-level representation of the compilation unit. The embedded IR will use the compiler's internal IR format so that the

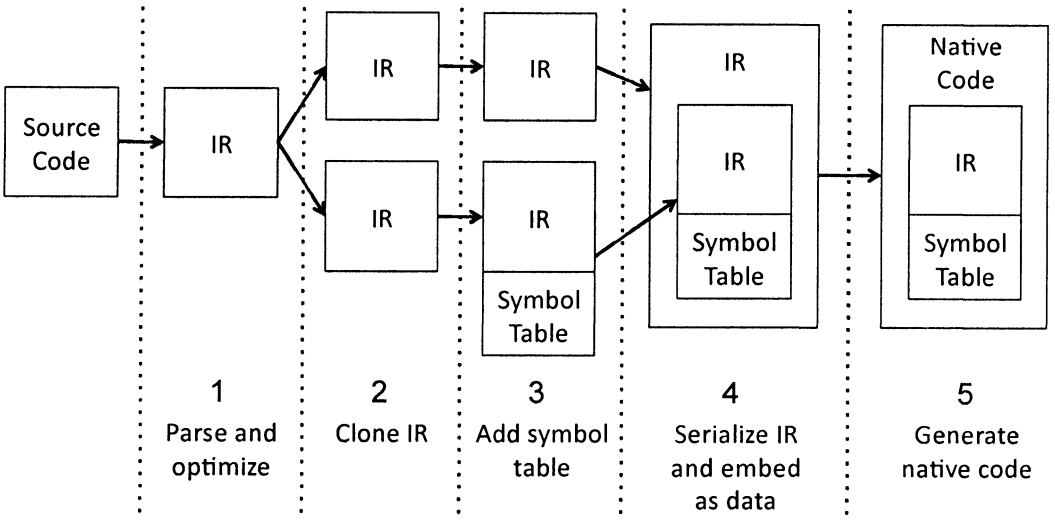


Figure 4.2: IR annotation workflow

existing compiler infrastructure can be leveraged for optimization and code generation at runtime. We use the LLVM Compiler Infrastructure for our implementation because it was designed from the start to support runtime code generation [33]. Tatge *et al.* describe a similar technique that is used for link-time optimization [49].

The main workflow for our IR annotation algorithm is outlined in Figure 4.2. The expected input is a program written in a statically compiled language; currently, we support C, C++ and Fortran. In step one we use the compiler to parse the program and convert it to the internal IR. Since all three supported languages are translated into the same IR, cross-language optimization will be possible with IR annotation.

Step two occurs immediately prior to the compiler's code-generation stage. We allow the compiler to apply all standard compiler passes so that we capture the program IR after it is fully optimized. By doing so, we avoid re-applying any redundant optimizations if we decide to re-compile the IR at runtime. For this step, we simply duplicate the program representation so that we have two identical copies. To achieve the best results, we recommend that both this step and code generation be delayed until program linking – this allows the linker to apply inter-procedural optimization across separate compilation units and to generate a single annotation. IR annotation

is still useful if applied to each compilation unit at compile-time, but it will result in separate annotations for each compilation unit.

For the IR to be useful for selective dynamic compilation, we must be able to resolve any referenced symbols in the module to their runtime address. This allows us to recompile a single function and link it in with the existing functions in the running application. We could use knowledge about the linker or object layout for this step, but that would decrease portability. Instead, we create, in the compiler's IR, a symbol table that will be embedded in the final application. Essentially, we are extending the program to contain a table of references to every unresolved symbol that we may need to know at runtime. Step three creates this symbol table by scanning the IR for all symbol references.

Step four generates a raw-data representation of the IR that can be embedded into the program's data section; we refer to this as *serialization.* We use LLVM's convenience functions for serializing the IR into a buffer; then, we embed the resulting data into the original program by inserting a constant byte array, initialized to contain the serialized IR. The symbol table is constructed as an array of pointers, initialized with an entry for each symbol that it contains. During this step we must also add an initialization and finalization function that will be called when this module is loaded into and unloaded from memory. These functions register and unregister the IR with the runtime system so that it knows which IR modules are available. The registration library function is very lightweight – it just builds a linked list of all the available IR modules in an application. To avoid dynamic memory allocation, we must also reserve space for linked-list pointers in each IR module.

Finally, step five generates native executable code for the original IR, which now has a serialized representation of the module embedded in the data section. When the linker is invoked it fills in the entries in the symbol tables and links in the runtime system library, producing an executable object. An application that is built from

several separate compilation units will contain an IR module for each; in this case, the final binary will simply contain multiple IR modules that will each be registered with the runtime system separately.

4.2.1 Runtime System

The minimum required runtime system is a small library that provides routines for registering IR modules as the program is loaded into memory. The registration routine just builds a linked list, using the statically allocated memory in each IR module for storing the list pointers. This runtime system is designed to be extremely fast; it only records the locations of the IR modules that are available in the program. This minimal IR annotation framework serves as a foundation for a full dynamic-compilation system, when extended with runtime performance monitoring and code generation—refer to Section 4.4 for more details.

4.3 Experimental Results

This section presents experimental results to verify the feasibility of IR annotation. The two main concerns with IR annotation are code growth and runtime overhead. We designed our runtime system to minimize runtime overhead, so we would expect the performance impact to be small or negligible—our experimental results confirm this. Code growth, on the other hand, cannot be avoided because the underlying mechanism for IR annotation is code duplication. However, our experimental results show that the code growth caused by IR annotation is reasonable.

For a general evaluation of IR annotation, we first considered the SPEC CPU2006 benchmarks. Figure 4.3 shows the performance impact from running an IR annotated benchmark with the minimal runtime system (see Section 4.2.1), as compared to running the same benchmark without IR annotation. The impact ranges from a

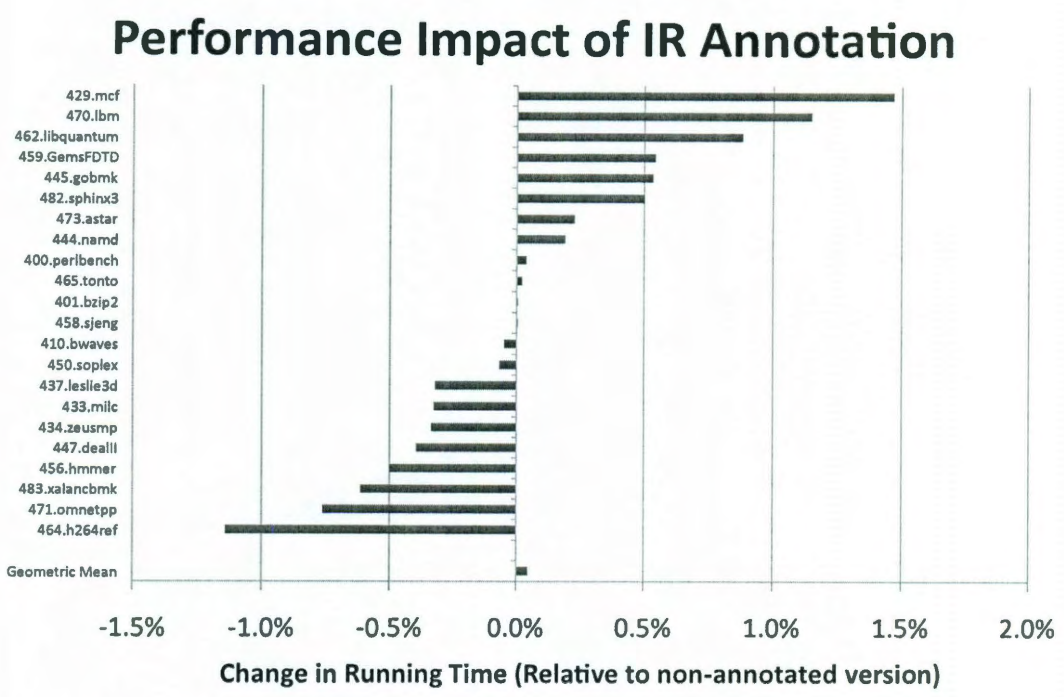


Figure 4.3: Performance impact for SPEC CPU2006 benchmarks

slowdown of approximately 1.5% to a speedup of just over 1%. The geometric mean is only about a 0.05% slowdown, suggesting that the expected impact of IR annotation is negligible. We do not claim that the observed speedups are benefits of IR annotation—rather, we speculate that the wide range in speedups and slowdowns can be attributed to other factors related to the object's layout in memory, a phenomenon similar to that seen in prior research [38]. Note that the runtime system in this experiment did not perform any runtime performance monitoring or dynamic compilation. Instead, the runtime system only registered the IR annotation for each functional unit, which involves storing a pointer in a linked list. The purpose of this experiment was to confirm that the base framework for IR annotation introduces insignificant overhead.

The code growth trend for the SPEC benchmarks is much more consistent, as shown in Figure 4.4. In all cases there is a positive code growth, from about 1.5x to just over 3x. The geometric mean indicates a code growth of just under 2.5x. A doubling in code size is understandable, since IR annotation places two representa-

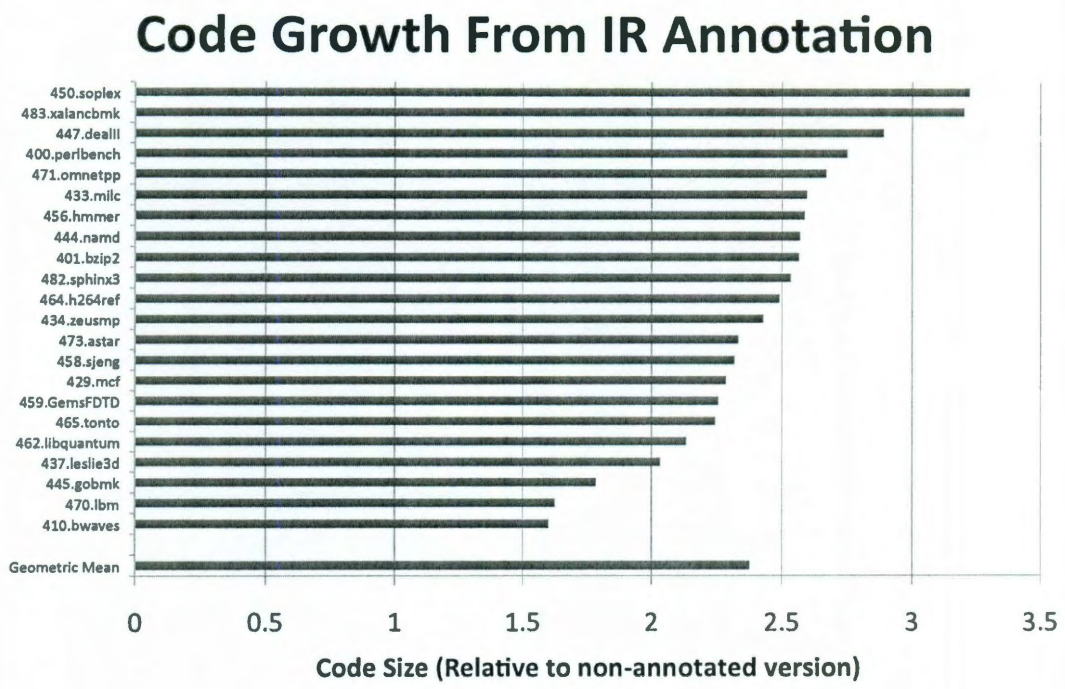


Figure 4.4: Code growth of SPEC CPU2006 benchmarks

tions of the program in the final executable: the native machine code and the IR. In practice, one might expect the code growth to be slightly larger than 2x, since it seems likely that the native machine code program representation is more compact than the compiler's IR. Also, some additional overhead can be attributed to the symbol table and initialization routines. Regardless, we believe the code growth caused by IR annotation is reasonable, especially since disk space is generally increasing in capacity and decreasing in cost.

Finally, we performed a case study of the CCA tools framework, version 0.7.0. Since the CCA is not an actual application—rather, it is a collection of tools and libraries upon which component-based applications can be built—we were unable to measure performance results. Nevertheless, we can still evaluate the CCA's code growth from IR annotation. Figure 4.5 shows the code growth for the executables and shared libraries that comprise the CCA tools, sorted by the size increase. Refer to Table 4.1 for the full listing of the corresponding file names. About half of the libraries have a 2x growth or less, while the other half range from 2-3x. The total size of the

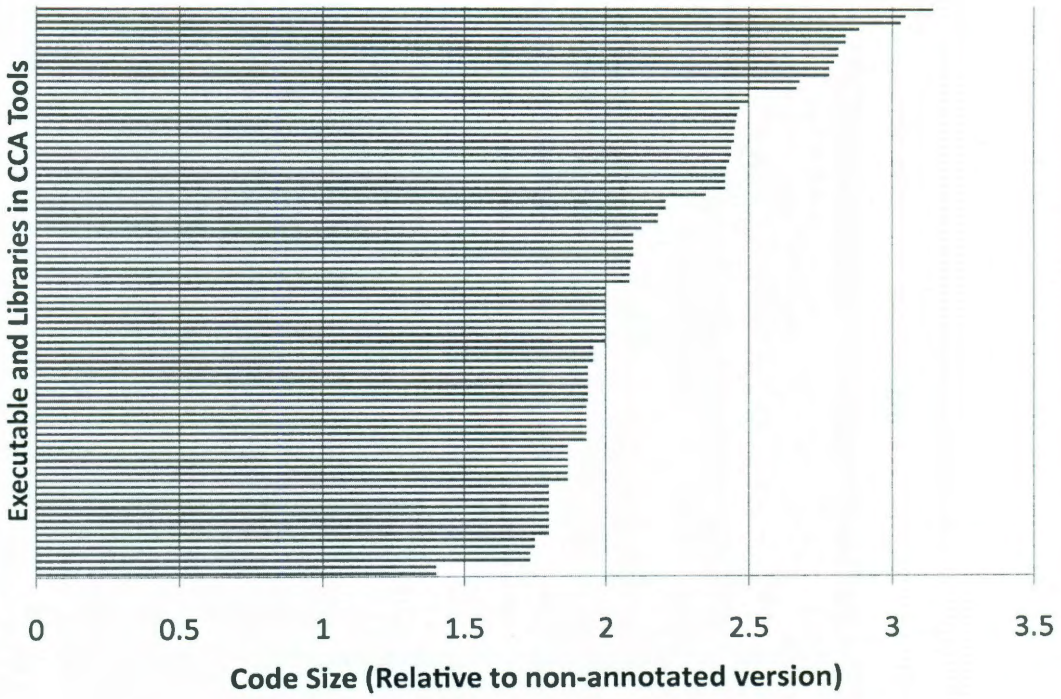


Figure 4.5: Code growth of binaries in CCA Tools

binaries resulting from standard compilation is 45 MB and the total size resulting from IR annotation is 114 MB, representing about a 2.5x increase. These results, which are very similar to the SPEC results, are not unreasonable for production use, especially considering the advantages of IR annotation. Additionally, the following section describes several improvements that may reduce the code growth caused by IR annotation.

4.4 Intended Use for IR Annotation

IR annotation, by itself, is of little use. Instead, it is a framework upon which a full dynamic compilation system can be implemented. Although implementing and evaluating such a system is beyond the scope of this work, we will briefly discuss our intended strategy for leveraging IR annotation.

We propose that IR annotation be used with a "do no harm" strategy. That is,

File	Growth
lib/libchasmlite-1.4.0.so	3.14
lib/libsidlstub_cxx-1.4.0.so	3.05
lib/libccaffeine_0_8_8.so.0.0.0	3.03
lib/libsidlstub_java-1.4.0.so	2.88
lib/libccafeCore_0_8_8.so.0.0.0	2.84
lib/libccafeDrivers_0_8_8.so.0.0.0	2.84
lib/cca-spec-babel-0_8_6-babel-1.4.0/libcca-cxx.so.0.0.0	2.81
lib/libcca_0_8_6_b_1.4.0-cxx.so.0.0.0	2.81
bin/cca-spec-babel-scanCCAxml.exe_0_8_6_b_1.4.0	2.80
lib/libcca_0_8_6_b_1.4.0-java.so.0.0.0	2.78
lib/cca-spec-babel-0_8_6-babel-1.4.0/libcca-java.so.0.0.0	2.78
lib/libsidl-1.4.0.so	2.68
lib/libsidlx-1.4.0.so	2.67
lib/libsidlstub_f77-1.4.0.so	2.50
lib/libSCPProxy_0_8_8.so.0.0.0	2.50
lib/libServiceRegistryTest_0_8_8.so.0.0.0	2.47
lib/libparsifal-1.0.0.so	2.46
lib/libConnectionEventServiceTest_0_8_8.so.0.0.0	2.46
lib/libGUIServiceTest_0_8_8.so.0.0.0	2.45
lib/libcca_0_8_6_b_1.4.0-f77.so.0.0.0	2.45
lib/cca-spec-babel-0_8_6-babel-1.4.0/libcca-f77.so.0.0.0	2.45
lib/libcca_0_8_6_b_1.4.0-c.so.0.0.0	2.44
lib/cca-spec-babel-0_8_6-babel-1.4.0/libcca-c.so.0.0.0	2.44
lib/libPrinterComponent_0_8_8.so.0.0.0	2.43
lib/libParameterPortFactoryTest_0_8_8.so.0.0.0	2.42
lib/libStarterComponent_0_8_8.so.0.0.0	2.42
lib/libGoComponent_0_8_8.so.0.0.0	2.42
lib/libSimpleProxyTest_0_8_8.so.0.0.0	2.42
lib/libBasicParameterPortTest_0_8_8.so.0.0.0	2.35
lib/python2.5/site-packages/gov/cca/ports/ParameterPort.so	2.21
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ParameterPort.so	2.21
lib/python2.5/site-packages/gov/cca/TypeMap.so	2.18
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/TypeMap.so	2.18
lib/libsidlstub_f90-1.4.0.so	2.13
lib/libcca_0_8_6_b_1.4.0-f90.so.0.0.0	2.10
lib/cca-spec-babel-0_8_6-babel-1.4.0/libcca-f90.so.0.0.0	2.10
lib/python2.5/site-packages/gov/cca/ports/BuilderService.so	2.10
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/BuilderService.so	2.10
lib/python2.5/site-packages/gov/cca/Services.so	2.09
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/Services.so	2.09
lib/python2.5/site-packages/gov/cca/ports/ParameterPortFactory.so	2.08
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ParameterPortFactory.so	2.08
lib/libStringConsumerPort_0_8_8.so.0.0.0	2.00
lib/python2.5/site-packages/gov/cca/Port.so	2.00
lib/python2.5/site-packages/gov/cca/AbstractFramework.so	2.00
lib/python2.5/site-packages/gov/cca/ConnectionID.so	2.00
lib/python2.5/site-packages/gov/cca/TypeMismatchException.so	2.00
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/Port.so	2.00
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/AbstractFramework.so	2.00
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ConnectionID.so	2.00
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/TypeMismatchException.so	2.00
bin/ccafe-batch_0_8_8	1.96
bin/ccafe-client_0_8_8	1.96
bin/ccafe-single_0_8_8	1.96
lib/python2.5/site-packages/gov/cca/CCAException.so	1.94
lib/python2.5/site-packages/gov/cca/ports/ServiceRegistry.so	1.94
lib/python2.5/site-packages/gov/cca/ports/BasicParameterPort.so	1.94
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/CCAException.so	1.94
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ServiceRegistry.so	1.94
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/BasicParameterPort.so	1.94
lib/python2.5/site-packages/gov/cca/ports/ServiceProvider.so	1.93
lib/python2.5/site-packages/gov/cca/ports/ConnectionEventService.so	1.93
lib/python2.5/site-packages/gov/cca/ports/ComponentRepository.so	1.93
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ServiceProvider.so	1.93
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ConnectionEventService.so	1.93
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ComponentRepository.so	1.93
lib/python2.5/site-packages/gov/cca/ComponentRelease.so	1.87
lib/python2.5/site-packages/gov/cca/ComponentID.so	1.87
lib/python2.5/site-packages/gov/cca/Component.so	1.87
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ComponentRelease.so	1.87
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ComponentID.so	1.87
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/Component.so	1.87
lib/python2.5/site-packages/gov/cca/ports/ParameterGetListener.so	1.80
lib/python2.5/site-packages/gov/cca/ports/ParameterSetListener.so	1.80
lib/python2.5/site-packages/gov/cca/ports/ConnectionEventListener.so	1.80
lib/python2.5/site-packages/gov/cca/ComponentClassDescription.so	1.80
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ParameterGetListener.so	1.80
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ParameterSetListener.so	1.80
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ConnectionEventListener.so	1.80
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ComponentClassDescription.so	1.80
lib/python2.5/site-packages/gov/cca/ports/ConnectionEvent.so	1.75
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/ConnectionEvent.so	1.75
lib/python2.5/site-packages/gov/cca/ports/GoPort.so	1.73
lib/cca-spec-babel-0_8_6-babel-1.4.0/python2.5/site-packages/gov/cca/ports/GoPort.so	1.73
lib/libccafePreload_0_8_8.so.0.0.0	1.40
lib/ccafe-0.8.8/libccafePreload.so.0.0.0	1.40

Table 4.1: Code growth of binaries in CCA Tools

the system should make every effort to avoid degrading performance for any program; but, in certain cases it may improve performance. Along these lines, we argue that any computation that can be performed statically (*i.e.*, ahead of time with respect to program execution) should never be performed dynamically (*i.e.*, during program execution). Thus, IR annotation should be combined with aggressive whole-program link-time-optimization to avoid performing optimizations dynamically that could have been performed statically. Then, any opportunities for optimization that arise at runtime should be unique to the runtime context. If such opportunities are present, then the dynamic compiler can selectively optimize only the relevant regions of code.

Upon program invocation, the minimal IR annotation runtime system, as described in Section 4.2.1, will collect pointers to all blocks of IR annotation in the program's executable and shared libraries. These pointers will be stored for later use, but the IR will not be de-serialized until it is needed—the goal is to delay as much overhead as possible until a potential benefit arises. Immediately prior to the invocation of the program's `main()` function, the runtime system should launch a light-weight performance-monitoring tool that will run in parallel with the running application. For best results we suggest using a sample-based profiler that leverages hardware performance counters, such as HPCToolkit [2]. A sample-based profiler has the advantage that profiling can easily be enabled and disabled. Also, the sampling overhead can be reduced by decreasing the sampling rate, at a cost of decreased profiling accuracy. The hardware performance counters provide rich detail beyond basic CPU cycles, often providing important insight into the location and cause of performance bottlenecks. For example, a particular loop or routine that consumes a large percentage of CPU time may not need optimization if it already achieves a system's peak instruction throughput.

The main idea behind light-weight performance profiling is that the performance cost is low enough that little harm will result even if no runtime optimization is per-

formed. However, if the profiling reveals particular optimization opportunities then the dynamic compiler can selectively target only those regions of code. Further, the compiler can leverage the profile information to direct which optimizations should be applied, potentially even determining the parameters of those optimizations. In this manner, the cost of invoking a dynamic compiler is never incurred if no opportunities are discovered. Yet, whenever the dynamic compiler is invoked, it is directly the result of discovering an opportunity. In this latter case, the benefit of invoking the compiler is much more likely to outweigh the cost of invoking it.

Although IR annotation is designed to be applicable to any existing compiler framework, we propose using LLVM because it was intended from the beginning to support runtime code generation. Thus, we implemented our IR annotation prototype in LLVM. Other compiler frameworks could certainly be used, but may require more effort to tailor them for use in a runtime context.

Once the profiler detects a region of code requiring optimization, the runtime system can launch the optimization thread. This thread is meant to run in parallel with the main application, allowing the main application to continue processing. The optimization thread can run at a lower priority than the application threads; alternatively, the main application could reserve a single hardware thread to which the optimization thread could be assigned. In either case, the performance impact of invoking the optimization thread should be minimal.

The optimization thread first needs to find the IR that relates to the region of code requiring optimization. This problem involves mapping addresses in optimized machine code back to the program structure. Tallent solves this problem generally, within the context of HPCToolkit, for mapping profiling information back to the program's original source code [48, 2]. We can use similar techniques, although our problem is easier because we only need to map back to the IR, which correlates much closer to the machine code than the original source code does. We can use the symbol

table, included with the IR annotation, to create a map from program addresses to IR modules. If a program is only partially annotated, then our map will contain "holes", indicating regions that we cannot re-optimize. Once the IR for the target routine is discovered, the compiler may then perform the desired optimizations on that code and any other relevant code. Note that these optimizations should be performed in a private "staging" area in memory that will not affect the running program's behavior.

Finally, once optimization is complete the runtime system should re-generate native code for all affected routines. The resulting code should be relocated into a special code cache from where it will execute. This model extends the program's original native code with the code cache, so multiple versions of some routines may exist. In fact, the main application will have continued executing the original code during the entire optimization process. The final step is to "patch" the original routine to include the new routine. This step is inherently low-level and platform specific, since we must overwrite the first instruction in each routine with a jump to the new routine. A more portable alternative is to use indirection tables, similar to program linkage tables in relocatable shared libraries. In this situation, a call to any routine first looks up the target in an indirection table and then performs an indirect jump. Although the indirection approach adds some extra performance overhead for each function call, the benefit is that patching a re-optimized routine into the running application becomes portable and trivial—the runtime system need only update an entry in the indirection table. In either case, the re-optimized routine will be executed when it is next invoked, and the original routine will remain unused for the rest of the program's execution.

After patching the program, the optimization thread can be temporarily suspended and the runtime system may return to its initial state in which it profiles the performance of the running application. The profiler should continue looking for other optimization opportunities, but it can also evaluate the impact of its previous

optimization decisions. In this manner the dynamic compiler may be invoked repeatedly to iteratively refine a particularly long running or dynamic application. As the code cache grows in size over time, the runtime system may reclaim space occupied by regions of code that have been since replaced by new versions.

There is one final consideration for our proposal of patching a running program with re-optimized routines. A patch only becomes active when the application reaches the point of entry, which in our model is the start of a routine. This approach works fine for short routines that are invoked many times. But, for long running routines that are invoked a small number of times, there may be an arbitrarily long delay from when the patch is installed to when the patch becomes active. Consider a `main()` routine that comprises an entire long-running, loop-based program—a patch to the `main()` routine will never become active, because `main()` is only called once per program execution. We cannot externally interrupt the executing thread and transfer control to the re-optimized routine without explicit information about the state of the program. This problem is very similar to one that arises in garbage collection for Java JIT compilers—when invoked, the garbage collector must be able to locate all object references on the heap or stack and in registers. This can be difficult when running in mixed mode, where some code is compiled and some is interpreted. Most JITs solve this problem by introducing *yield points* or *safe points* at which the garbage collector can be called [6]. At such points, the runtime system provides a *reference map* that indicates the location of object references. The goal of such safe points is to ensure that the time between calls to the garbage collector will be bounded by a finite number of steps. Similarly, we could employ a version of safe points for IR annotation that place an upper bound on the time that may elapse between a patch's application and subsequent activation. Such a safe point would provide a location in which we could safely interrupt a thread and transfer control to a new version of the current routine. Perhaps we could leverage existing technologies, such as C++

exception handling, to implement safe points in a portable and low-overhead manner.

4.5 Related Work

Dynamic compilation has been widely used with object-oriented languages, such as Smalltalk [19], Self [13], and Java [5, 12, 39, 47]. These languages are generally statically compiled into non-native bytecode, which must then be interpreted with a virtual machine. The primary benefit of dynamic compilation for these systems is that frequently interpreted code can be translated into native code, which is almost always guaranteed to run faster. In other words, dynamic compilation is profitable if a function or loop body will execute long enough to make up for the cost of invoking the compiler. Optimization opportunities are identified simply by measuring function and loop execution counts and comparing against a pre-defined threshold. IR annotation, on the other hand, already starts with optimized native code; thus, dynamic compilation is certainly not guaranteed to improve performance. In fact, with no new information, the compiler will likely re-generate equivalent code. Consequently, IR annotation requires that dynamic compilation be applied much more selectively. Frequent execution alone is not a sufficient trigger for dynamic compilation—instead, a true runtime optimization opportunity should be first discovered.

The Jalapeno compiler for Java is noteworthy because it was able to generate code both statically, ahead of time, and dynamically, as a just-in-time (JIT) compiler [12]. It appears that the expected use of this functionality was for bootstrapping the virtual machine (since it was written in Java), or for fully optimizing a program offline. IR annotation is intended to optimize and generate code offline, yet still allow dynamic re-compilation.

Research on binary optimization is also relevant to IR annotation. The Dynamo system by Bala *et al.* [8] identified and optimized hot traces in the stream of native

instructions in a running program. This novel approach enabled optimization across procedure and shared-library boundaries, regardless of the language in which code was originally written. IR annotation also aims to allow optimization across such boundaries, but it is only possible if both libraries are annotated with IR. Dynamo's overhead was sometimes significant because it used interpretation to profile the running program. Lu *et al.* addressed this problem in their ADORE system by using lightweight sample-based instrumentation and selectively applying optimization only on regions of code with a potential optimization opportunity [34]. Our approach more closely matches that of Lu *et al.* because we also propose sample-based profiling and selective optimization, but we do not propose binary optimization. Although binary optimization systems can target any programming language, they are very low-level and inherently platform-dependent. IR annotation allows the dynamic compiler framework to largely remain above such low-level details by shifting the platform-specific information into the dynamic compiler's backend. This separation of concerns improves portability, allows the compiler to leverage a mature and existing platform-independent compiler infrastructure, and enables higher-level optimization.

Finally, Pin is binary instrumentation system that makes effective use of binary optimization to reduce the overhead of instruction-level instrumentation [35]. The main benefits in Pin's optimizer come from inlining user-specified instrumentation code into the main program stream and optimizing for the context. Although Pin is very effective at reducing the cost of dynamically instrumenting a running program, it is not designed to improve the performance of non-instrumented programs.

4.6 Conclusion

While component-based programming promises to improve the productivity of HPC developers, it must be accompanied by compilers and tools that can deliver the per-

formance that the HPC community has come to expect. In this chapter we argued for using dynamic compilation to reduce overhead in component-based applications. We described IR annotation, an approach for enabling selective dynamic compilation for statically compiled languages. Experimental results confirm that our lightweight runtime system results in negligible performance overhead for IR annotation. And, although IR annotation does result in a size increase for program binaries, our experiments show that the growth is only about 2.5x.

More generally, IR annotation is a foundational technology upon which a full dynamic compilation system can be built. Chapter 5 presents adaptive code-selection, a dynamic optimization technique that can be paired with IR annotation for dynamically performing feedback-directed optimization. Just as resource characterization will allow a static compiler to automatically adapt to rapid changes in computer hardware, IR annotation will allow a dynamic compiler to adapt to unpredictable or unexpected behavior of software applications. These two technologies complement one another in the goal of de-coupling a program's source code from the data that it will process and the hardware upon which it will run.

Chapter 5

Adaptive Code Selection

Traditionally, adaptive compilation, also called iterative compilation or empirical tuning, uses an offline compile-execute-adapt feedback loop. An adaptive compiler optimizes a program and generates code, then runs the program with training-data input. The performance information is recorded and fed back to an adapter, which modifies the optimization decisions used in the compile step for the next iteration. Given a sufficient number of iterations, the adaptive compiler often produces better code than the traditional static optimization approach, by tuning the optimization decisions for a particular program and input set. Although the number of iterations needed when using random probing can be quite high (thousands or tens-of-thousands), artificial-intelligence search techniques (such as hill climbers or genetic algorithms) can significantly reduce the time to find a better solution [15, 28, 45, 50]. The main drawback of this form of adaptive compilation is that it tunes an application to perform well for a particular training input. If that input is not representative of the manner in which the application will actually be used, then the optimization may not help performance (and it may actually hurt performance). Also, the tuning setup requires programmer intervention to specify the training input data and parameters. Instead, if we apply adaptive compilation at runtime, within the context of a single program execution,

then tuning can be much more targeted because the adapter will receive feedback from the currently running application. The distinction between the training and real-world input sets disappears. Additionally, applying adaptive compilation at runtime eliminates the burden on the developer to specify input parameters and explicitly invoke the offline adaptive compiler—the runtime system can adaptively optimize an application without the developer's intervention, or even knowledge. However, it is widely recognized that the costs of this offline technique are quite high. This section describes how to modify adaptive compilation to be effective within the context of a single application invocation. We first describe the program constructs and behaviors that we believe are amenable to runtime adaptive compilation, and then outline the implementation details. Finally, we conclude by presenting results from a case study that explores the practical benefits of this technique.

5.1 Overview

Traditionally, offline adaptive compilation repeatedly executes an entire application, tuning the optimization decisions between executions. An online version of this technique must shift the adaptive feedback loop so that it occurs within a single program execution. The offline version introduces artificial program-repetition by re-executing the entire application once per iteration. In contrast, the online version will operate within normal program execution—instead of introducing artificial repetition, it must exploit the repetition that naturally occurs in the program stream. Thus, this technique applies to heavily repeated regions of code, either loops or functions. We refer to this heavily repeated region of code as the *kernel*.

Early during the program's execution, the runtime system should begin generating different versions of the kernel with various *optimization sequences*, or ordered sequences of optimization decisions. Each different version will be referred to as an

instance of the kernel, or *kernel instance.* As the program executes, the runtime system should exercise the different kernel instances, by dynamically swapping them in and out of the program's stream of execution. The runtime system should collect performance information to evaluate the performance of each kernel instance, and therefore the effectiveness of each optimization sequence. After a sufficient evaluation has been made, the dynamic compiler can tune the optimization sequences based on the performance feedback and generate new kernel instances. Over time the runtime system should begin to identify certain kernel instances that out-perform the others; the slower instances can be gradually withdrawn from use while the faster instances can be allocated more execution time. This evaluation and tuning process can repeat indefinitely; or it can phase out after the observed improvements diminish, at which point the fastest kernel instance should be employed for the remainder of program execution.

Since this approach requires that the dynamic compiler be invoked many times (once for each kernel instance), the runtime overhead could be significant. Certainly, if the program does not run long enough, or if the runtime system is unable to produce faster kernel instances, then the overhead of this technique may outweigh the benefits, resulting in an overall program slowdown. Therefore, this technique should be applied selectively, only when the program's actual execution behavior establishes that a region of code is heavily repeated. Additionally, the dynamic compiler must carefully select the optimization sequences that it tests, to avoid wasting time evaluating sequences that are unlikely to improve the results. If the runtime system is able to produce and identify kernel instances that out-perform the original, statically-generated code, and the kernel repeats enough, then this technique should result in an overall program speedup.

An example will help clarify this technique. Consider an application that repeatedly calls some function, `foo()`, which performs a single unit of work in the program's

main computation. Overall, the application spends a significant amount of time executing in `foo`, spread across a large number of separate invocations. This function is a prime candidate for runtime adaptive optimization; thus, we select `foo` as our kernel. The code may look like this:

```
void foo(int i) {
  /* Perform a unit of work ... */
}
void compute() {
  int i;
  /* Read input ... */
  for (i = 0 ; i < N ; i++) {
    foo(i);
  }
  /* Write output ...*/
}
```

To apply runtime adaptive compilation, the runtime system should intercept program control each time `foo` is called. Then, instead of executing the default implementation of `foo`, the runtime system can pass control to various dynamically-generated instances of `foo`. The runtime system acts as a wrapper for `foo`, allowing it exercise a different instance of `foo` upon each invocation. Additionally, the runtime system can measure the duration of each invocation of `foo` and gradually begin to identify and favor the better instances. The transformed version of `foo` may look like this:

```
void foo_orig(int i) {
  /* Perform a unit of work ... */
}
void foo(int i) {
  /* Possibly generate new version of foo */
  /* Call and time some version of foo */
  /* Review statistics and adjust future actions */
}
```

Notice that the original version of `foo` is preserved, to allow a fall-back in case the adaptive compiler is unable to improve performance. The mechanics of the adaptive optimization technique are contained entirely within the new version of `foo`, but the implementation is carefully designed to minimize the average overhead. The following sections provide more specific details.

5.2 Requirements and Pre-conditioning

There are several requirements that must be met before runtime adaptive compilation can be employed, and the code must be transformed and pre-conditioned ahead of time. This section describes these requirements and steps in more detail.

First, the compiler must identify kernel candidates—heavily repeated regions of code, such as long running loops or frequently executed functions are suitable candidates. Since adaptive compilation may incur significant overheads, the kernel must repeat enough times to amortize the overhead with respect to the benefit. Also, to allow different versions of the kernel to be "plugged in", the kernel should conform to a consistent programming interface. Functions explicitly conform to the interface defined by their function signature and the system's calling convention. Loop iterations have a less-explicit interface, which is defined by the pre-conditions and post-conditions of the entry and exit blocks of the loop body. For convenience in our discussion and implementation, we assume that a kernel is always a function rather than a loop body. This assumption does not limit generality, since a loop body can always be extracted into a new function and replaced by a function call, a transformation similar to the one proposed by Hall *et al.* [29]. We define a *kernel invocation* as a single execution of the kernel code.

Next, the runtime system must be capable of objectively comparing the performance between two different kernel invocations. In the simple case, when each invocation performs the same amount of work, execution time (or cycle count) is the ideal metric—this allows an "apples-to-apples" comparison between the running time of different invocations. If each invocation may perform a different amount of work (*e.g.,* a triangular or other irregular loop structure), then a comparison is much more difficult because an observed difference in running time might be attributed to the difference in work (instead of differences in performance). There are several potential solutions for this problem. First, if the irregular kernel is large enough then it may

be possible to sub-divide it into smaller equally-sized units of work. For example, a triangular loop-nest could be tiled and transformed so that the kernel executes a single tile.[1] After this transformation, the kernel is guaranteed to perform the same amount of work upon each invocation. An alternative option may be to normalize the measured execution time based on the amount of work the invocation performs, allowing a comparison between different invocations. For example, computations per second (or transactions per second) could allow for a fair comparison between two invocations that perform different amounts of work. This approach needs to quantify the amount of work, which may be difficult to do automatically. Finally, it may be possible to allow each kernel to perform different amounts of work. If the amount of work per invocation varies, but is uniformly distributed, then we may be able to rely on the statistical average over a large number of samples. Given enough samples, we could ignore differences in the actual amount of work performed per invocation, because the average amount of work should be roughly equivalent. However, this thesis will only consider the simple case in which the amount of work per kernel invocation is fixed; normalized metrics and statistical averaging are left for future work.

5.3 Implementation

The compiler can enable runtime adaptive compilation of a kernel with a few simple steps. First, if the kernel is a loop body then it should be extracted into a self-contained function. Since such an extraction could hide some important optimization context, we should perform all aggressive optimizations prior to extracting the kernel. Once the kernel code is extracted, we can replace the original code with a call to a control function that invokes the adaptive compiler at runtime.

Then the call site that calls the kernel should be replaced with a call to a control

[1]The edge cases that do not fill an entire tile can simply be excluded from the kernel and left to execute as normal.

function in the adaptive-optimization runtime-system. Since the control function is invoked in place of the actual kernel, it must implement the original kernel interface—in the object-oriented design-pattern paradigm, the control function acts as an adapter or wrapper for the kernel function . Thus, upon every invocation the control function will execute exactly one call to a kernel instance. Additionally, the control function performs other bookkeeping tasks for the adaptive optimization algorithm, both before and after the kernel invocation. For example, the first several calls to the control function should establish a performance baseline by invoking the original kernel code, which was generated by the standard optimization sequence. The control function measures the duration of each kernel invocation and maintains running statistics, such as the overall mean, standard deviation, and moving average. Once a baseline is established, the control function can begin generating and invoking new instances of the kernel using different optimization sequences. Two kernel instances should not be compared after a single invocation of each, because timing fluctuations may lead to an incorrect conclusion about which kernel is faster. Instead, the control function should invoke each kernel instance multiple times to establish average running times. Then, the performance metrics for each optimization sequence can be provided to the adapter for generating the next optimization sequence, completing the feedback loop. Once a set amount of time has passed, or the adaptive compiler identifies an optimization sequence that outperforms the default sequence, then the compiler can stop generating new sequences and use the best known sequence. After this point, each subsequent call to the control function will transfer directly to the kernel implementation.

If the duration of the kernel instances is too low to amortize the overhead of the control function, then the control function can always bail-out and enter a fall-back state to minimize future overhead. At this point, the adaptive optimization algorithm is terminated and each call to the control function will directly invoke the original

kernel.

5.4 Case Study

This section presents a small case study that explores the potential benefit of runtime adaptive optimization on matrix-matrix multiplication. Matrix-matrix multiplication was selected because it is important in many scientific algorithms and it contains three perfectly nested loops that expose significant repetition. The algorithm is also small, easily understood, and well studied.

This case study evaluates the effectiveness of the adaptive-selection algorithm, given a set of kernel instances. The kernels are compiled statically to simplify the experimental setup. The adaptive kernel selection algorithm is hand coded for the matrix-matrix-multiply example, but this should be straightforward to automate for more general application. To evaluate the overall technique of runtime adaptive optimization, we estimate the overhead for dynamically compiling the kernels. The final analysis presents results both with and without these estimates.

5.4.1 Static Code Transformations

The original matrix-matrix multiplication code is shown in Figure 5.1. Each of the three main matrix-matrix-multiply loops has been tiled for improving cache performance—the inner three loops iterate over a single tile while the outer three loops iterate over the set of tiles. The tile sizes are specified as parameters to the function, allowing runtime specification of tile sizes. This code is meant to represent a standard implementation of the matrix-matrix multiply algorithm.

For the purposes of this case study we will assume that the main program will only perform a single matrix-matrix-multiplication operation.[2] This means that the

[2]In practice, scientific applications that employ matrix-matrix multiplication usually invoke the routine many times during each execution. This repetition offers further opportunity for runtime

```
void matrix_multiply(int M, int N, int K,
                     int Ti, int Tj, int Tk,
                     double beta, double alpha,
                     double **A, double **B, double **C)
{
  int i, j, k;
  int it, jt, kt;

  for(kt=0; kt<K; kt+=Tk)
    for(it=0; it<M; it+=Ti)
      for(jt=0; jt<N; jt+=Tj)
        for(k=kt; k<min(kt+Tk, K); k++)
          for(i=it; i<min(it+Ti, M); i++)
            for(j=jt; j<min(jt+Tj, N); j++)
              C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];
}
```

Figure 5.1: Original Matrix-Matrix Multiply Code

```
void matrix_multiply(int M, int N, int K, int Ti, int Tj, int Tk,
                     double beta, double alpha,
                     double **A, double **B, double **C)
{
  int i, j, k, it, jt, kt;
  for(kt=0; kt<K; kt+=Tk)
    for(it=0; it<M; it+=Ti)
      for(jt=0; jt<N; jt+=Tj)
        inner_loop(it, min(it+Ti, M),
                   jt, min(jt+Tj, N),
                   kt, min(kt+Tk, K),
                   alpha, beta, A, B, C);
}

void inner_loop(int it, int ilim, int jt, int jlim, int kt, int klim,
                double alpha, double beta,
                double **A, double **B, double **C)
{
  int i, j, k;
  for(k=kt; k<klim; k++)
    for(i=it; i<ilim; i++)
      for(j=jt; j<jlim; j++)
        C[i][j] = beta*C[i][j] + alpha*A[i][k] * B[k][j];
}
```

Figure 5.2: Matrix-Matrix Multiply Code with Extracted Kernel

`matrix_multiply` function will only be invoked once per execution, so it cannot be used as the target of the adaptive optimization. The nested loops, on the other hand, offer significant repetition. We can create the kernel by extracting some subset of the inner loops into a new function. This extraction defines the granularity on which the adaptive optimization will occur—extracting only the innermost loop maximizes the kernel's iteration count, but minimizes its running time. In the other extreme, extracting all but the outermost loop maximizes the kernel's running time, but minimizes its iteration count. Ideally, we want to maximize the kernel's iteration count (to maximize the opportunity for adaptive optimization), while constraining the kernel's running time to be long enough to amortize the framework overhead. In Section we experimentally estimate the lower bound on a kernel's running time. For now, we will create the kernel by extracting the innermost three loops, as shown in Figure 5.2.[3] This choice of granularity balances a large number of calls to the kernel with a non-trivial running time for each kernel (as determined by the tile sizes). After the transformation, the inner three loops become the kernel, which is invoked once per tile.

Although the kernel extraction is a simple transformation, it is possible that it may impose a significant performance penalty. We would like to quantify the performance impact of the extraction to ensure that the overhead is not unreasonable. Since kernel extraction can be modeled as the inverse of function inlining, its cost should be roughly equivalent to the benefits of inlining. Thus, we should expect at least two sources of performance overhead: (1) the cost of the introduced function call and calling convention and (2) the optimization opportunity-cost of removing the kernel from its surrounding context. If the kernel runs long enough, we would expect to amortize the first type of overhead. In theory, we can also minimize the impact

adaptation and overhead amortization that our case study does not exploit.

[3]For this experiment we also placed the extracted function into a separate file; doing so prevents the compiler from re-inlining the kernel.

of the optimization opportunity-cost by performing the kernel extraction only after applying the standard compiler optimizations to the original code. By performing kernel extraction last, the compiler is allowed to exploit any context-based optimization opportunities. However, since we are performing the kernel extraction manually at the source code level, before the compiler has performed any optimization, we may experience an optimization opportunity-cost overhead. Fortunately, our experimental results, shown in Figure 5.3, indicate that the kernel extraction does not introduce any overhead. All running times, shown on the x-axis, are normalized to the time taken by the original code for each matrix size; the matrix sizes are shown on the y-axis. The performance of the original code is shown by the bars labeled "Original" and the performance of the matrix-matrix multiply with an extracted kernel is shown by the bars labeled "Extracted". Interestingly, for this code and this particular compiler, the version of matrix-matrix multiply with the extracted kernel actually runs faster than the original non-extracted version. Kernel extraction can improve performance for the same reasons that function inlining can degrade performance. Improved register allocation is a likely explanation for the results in Figure 5.3, becuase the function call in the middle of the loop nest forces register spilling at a better location than for the full loop nest.

5.4.2 Adaptive Kernel Selection

The main goal of adaptive kernel selection algorithm is to empirically determine the *best* kernel from a set of candidate kernels, and to favor that kernel for future invocations. Additionally, this evaluation and selection of a kernel should be accomplished within the application's natural stream of execution. That is, invocation and evaluation of a kernel can only be triggered when the application places an actual call to the kernel. Once the call is placed, however, the adaptive kernel-selector is allowed to invoke any kernel implementation. Thus, the adaptive optimization will occur, over

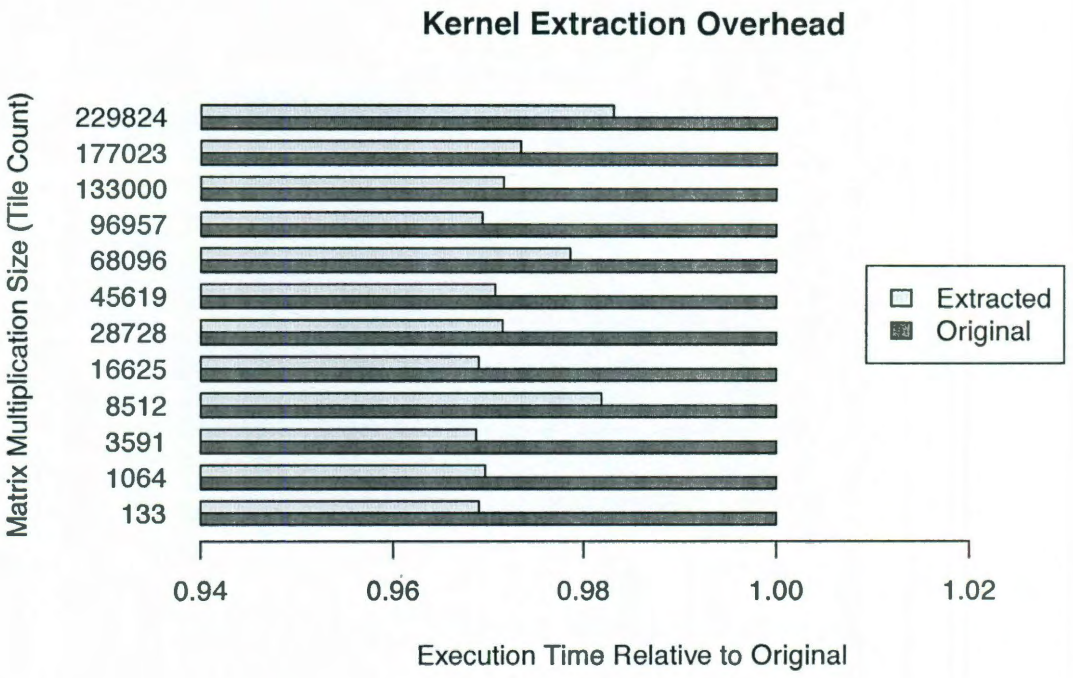


Figure 5.3: Kernel Extraction Overhead

time, as the application invokes the kernel control function.

We define best as fastest, but the algorithm could certainly select kernels that optimize other metrics, such as cache misses or power consumption. The only requirement is that there exist some mechanism for measuring that metric. This case study measures the elapsed time, in cycles.

In practice, the adaptive kernel-selection algorithm should compile the kernel instances at runtime to allow for proper feedback-directed adaptation. For this case study, however, we compile the kernels statically and only exercise the adaptive kernel-selection algorithm at runtime. This allows us to evaluate the ability of the kernel-selection algorithm to detect and favor the faster kernels over the slower kernels. In the final analysis, we estimate the overhead that would be incurred for dynamically compiling the kernel instances.

Our case study focuses on selecting an appropriate unroll factor for the innermost loop in the matrix-matrix-multiply kernel. We generate a set of 32 kernels, which are variations of the original kernel with unroll factors from 1 to 32. We unrolled the loop by hand for each instance, because of limitations with the LLVM loop unroller—LLVM's loop unroller does not handle symbolic loop bounds appropriately, and it also unrolls outer loops if the code growth remains within a threshold. After unrolling, we applied the standard "-O3" optimizations to each kernel. Since an unroll factor of one indicates no unrolling, the first kernel represents only the standard optimization.

Although it may be possible to statically determine effective unroll-factors, loop unrolling is still a suitable example for our case study. Dasgupta [18] showed that selecting an unroll factor at runtime, given a program's execution context, can be much more effective than selecting an unroll factor at compile time. Dasgupta's results suggest that loop unrolling is a good candidate for adaptive code-selection. However, even if optimal static loop-unrolling were possible, it would not detract from our technique—adaptive code-selection is still of value for other, less predictable,

optimizations.

To ensure that loop unrolling actually affects performance we individually exercised each kernel to determine its average performance. Figure 5.4 shows the results for the matrix-matrix-multiplication code when each kernel is used exclusively. The vertical line at 1.0 shows the performance of the original kernel, with standard optimization and no unrolling, and the horizontal bars show the performance of the kernels with unroll factors from 2 to 32. Clearly, many of the unrolled versions of the kernel perform much worse than the standard kernel—on average, the unrolled kernels perform over 7% slower than the standard kernel. However, kernels with unroll factors of less than 6 perform slightly better than the standard kernel, with unroll factors of 2 and 3 performing the best. It can be difficult to statically predict this result—that is, the static compiler can't always model and predict the most effective unroll factor. However, runtime adaptive optimization will allow us to empirically determine the best unroll factor at runtime.

The adaptive kernel selection algorithm works as follows. Upon the first call to the kernel interface, the set of kernel instances is initialized. In practice, this might be where the dynamic compiler actually generates the set of kernel instances. For our case study, the instances are already compiled, so we just construct an array of kernel pointers. We also initialize a data structure for maintaining running statistics for each kernel instance; the algorithm incrementally computes the number of invocations of the kernel and the sum of the invocation durations, which are used to compute the average invocation duration. Each kernel instance is also assigned an allocation weight, which determines the fraction of invocations that will be serviced by that kernel during each *epoch*. An epoch is a period of time during which the kernels are exercised, for a specified number of kernel invocations. At the end of each epoch the kernel weights are readjusted. This case study starts with an epoch size of 1000 iterations but allows for increasing the epoch size if one kernel establishes itself as

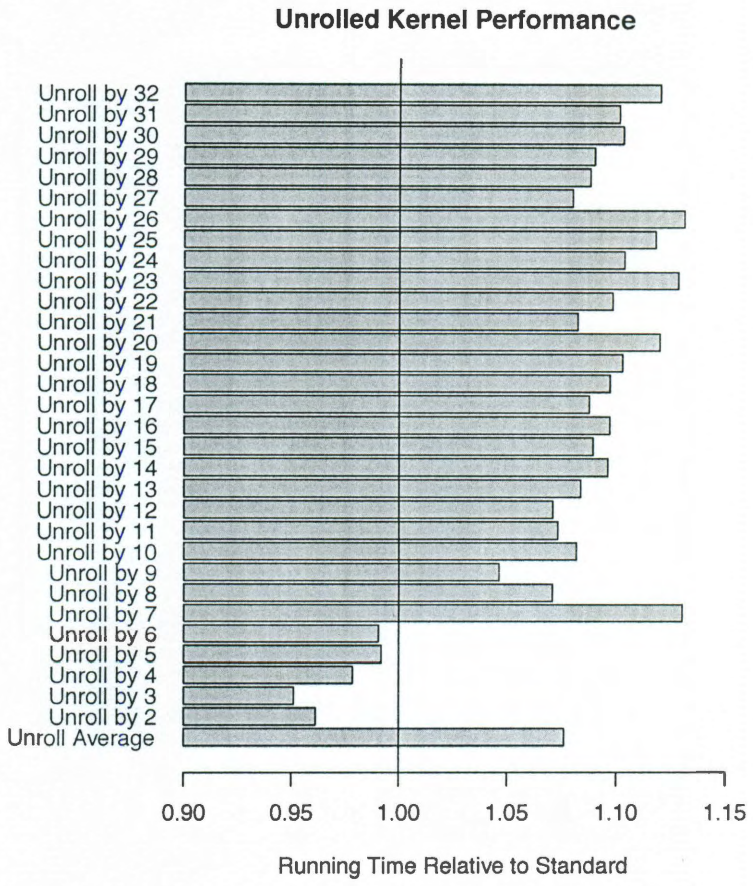


Figure 5.4: Unrolled Kernel Performance

```
void inner_loop(int it, int ilim, int jt, int jlim, int kt, int klim,
                double alpha, double beta,
                double **A, double **B, double **C)
{
  /* Find the next handler with remaining iterations */
  while (kernels[index].remainingIterations <= 0) {
    index++;
    if (index >= kernelCount) {
      adjustAllocation();
    }
  }
  /* Decrement the remaining iterations for this kernel */
  (kernels[index].remainingIterations)--;

  /* Invoke and time this kernel */
  uint64_t start = rdtsc();
  kernels[index].functionPointer(it, ilim, jt, jlim, kt, klim,
                                 alpha, beta, A, B, C);
  uint64_t stop = rdtsc();

  /* Update the running statistics for this invocation */
  updateSample(index, (double)(stop - start));
}
```

Figure 5.5: Adaptive Selection Kernel Control Function

the fastest option. For the first epoch, the kernel allocation weights are assigned to equally divide the epoch between all of the kernels.

Upon each invocation, the control function executes the current kernel, specified by the global `index` variable, as long as that kernel has one or more iterations remaining in the current epoch. The control-function code is shown in Figure 5.5. When the current kernel has no iterations remaining, then the `index` counter is incremented to begin exercising the next kernel. When the `index` counter moves past the last kernel, the epoch is over. The kernel weights are readjusted and the `index` counter is reset to zero to begin the next epoch. Upon each invocation, the kernel's duration is measured and recorded in the running-statistics data structure for that kernel—these statistics are used to adjust kernel allocation weights in between epochs.

```
const double iterationAlpha = 0.50;
const double winnerAlloc = 0.90;

void computeNewIt(int i, double baseline, double scale,
                  double weight, double bonusAlloc)
{
  double targetAlloc = (weight*((baseline-handlers[i].getMean())*scale) +
                        bonusAlloc);
  double newAlloc = (iterationAlpha*(targetAlloc) +
                     (1.0-iterationAlpha)*handlers[i].alloc);
  handlers[i].totIt = (uint64_t)(newAlloc*(double)epochSize);
  handlers[i].alloc = newAlloc;
  handlers[i].remIt = handlers[i].totIt;
}

void readjustAllocation() {
  /* Reset index */
  index = 0;

  /* Find min, max, sum, and count of kernel averages */
  double epochMin = ...;
  double epochMax = ...;
  double epochSum = ...;
  uint64_t epochCount = ...;

  /* Find index of epoch winner */
  int winnerIndex = ...;

  /* Adjust epoch size */
  if (handlers[winnerIndex].totIt * 2 > epochSize) {
    epochSize = std::min(maxEpochSize, epochSize * 2);
  } else {
    epochSize = std::max(minEpochSize, epochSize / 2);
  }

  double baseline = epochMax;
  double scale = 1.0 / (epochMax*(double)epochCount - epochSum);

  /* Compute new allocation and iteration for each kernel */
  for (int i = 0 ; i < winnerIndex ; i++) {
    double bonus = (i == winnerIndex) ? winneAlloc : 0.0;
    computeNewIt(i, baseline, scale, 1.0 - winnerAlloc, bonus);
  }
}
```

Figure 5.6: Adaptive Kernel Allocation

5.4.3 Reallocating Iteration Weights

At the end of the first epoch, each kernel will have been exercised an equal number of iterations. The running statistics for each kernel will indicate the number of invocations and the sum of durations for those invocations. The reallocation algorithm is shown in Figure 5.6. First, the algorithm computes the overall average duration for each kernel. Then, the kernel with the fastest average duration is selected as the epoch winner, and is automatically assigned a winner's bonus for the next epoch's allocation. We used a 90% bonus for this case study—the epoch winner automatically receives a target of 90% of the iterations for the next kernel, while the remaining 10% of the iterations are distributed among the other kernels. The distribution linearly allocates iterations to the kernels based on their running times; the slowest kernel receives a target allocation of zero while the middle kernels receive some fraction. Finally, to prevent drastic changes to the kernel allocation, the algorithm computes the new allocation as a weighted sum of the old allocation and the target allocation. The case study uses an alpha of 0.5, which means that the new allocation is computed as the arithmetic mean of the old allocation and the target allocation. Over time, if one kernel continually remains the epoch winner, then its actual allocation will eventually reach the 90% target allocation. If the epoch winner changes after each epoch, then several kernels will compete for a fraction of the 90% winner's bonus. The alpha factor determines how fast the algorithm will allow the dominant kernel to change.

Finally, during the reallocation of weights the epoch size is allowed to be increased or decreased, depending on the recent results. If the current winner already has an allocation above some threshold, 0.75 for the case study, then the epoch size is doubled (up to a pre-defined maximum of 10,000). This allows the algorithm to recognize that a fast kernel has been found, and that reallocation should occur less often in an attempt to further reduce the overhead. However, if the current epoch winner has an allocation below the threshold, then the epoch size is halved (down to

a pre-defined minimum of 1,000). This allows the algorithm to readjust the allocation more frequently if no consistent epoch winner emerges.

5.4.4 Results

Figure 5.7 shows the results of the adaptive-selection algorithm, as compared to the individual kernel performance. The adaptive selection algorithm, specified by "Adaptive", achieves roughly a 4% speedup over the "Standard" kernel. Since the adaptive kernel-selection algorithm must execute all kernel instances equally during the first epoch, to determine which one is the fastest, we cannot expect it to match the performance of the best kernel. However, the results indicate that the efficiency of the kernel selection algorithm is quite good, since its performance is very close to the best two kernels. More importantly, the adaptive performance significantly outperforms the average performance of the unrolled kernels—this indicates that the adaptive algorithm is correctly identifying and favoring the fastest kernels.

Figure 5.7 shows the performance for a single, very large matrix size. The long compute-time of the large input creates favorable conditions for the adaptive-selection algorithm, because it allows the overhead to be amortized. Next, we'd like to examine the adaptive algorithm's efficiency for a wide variety of matrix sizes to determine the "break-even" point—Figure 5.8 presents these results. The "Adaptive" bars indicate the percent speedup, relative to standard optimization, that the adaptive selection algorithm achieves for various matrix sizes. The "Optimal" bars indicate the speedup achieved by the fastest individual kernel for each particular size—these bars estimate an upper bound on the speedup that the adaptive selection algorithm can achieve.[4] For the larger matrix sizes, on the right side of the graph, the adaptive-selection

[4]It is possible for the adaptive-selection algorithm to perform better than the optimal individual-kernel, if the application or system exhibits changes in context or behavior that favor different kernel choices at different times. Since matrix-matrix multiply does not exhibit such a behavior change, we would only expect the adaptive-selection algorithm to outperform the optimal individual-kernel if the system environment changes during execution.

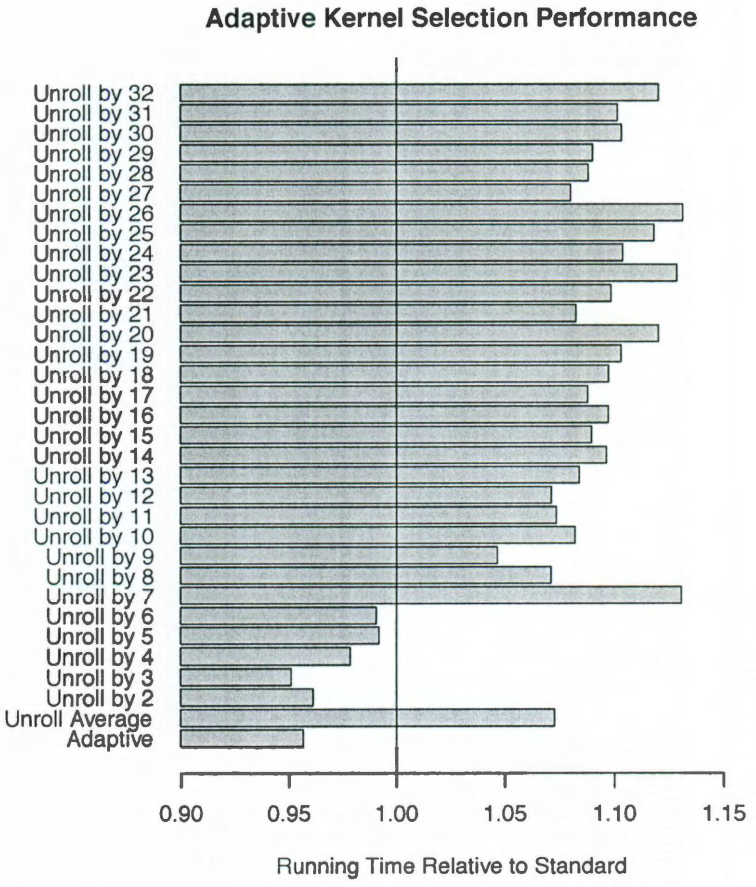


Figure 5.7: Adaptive Kernel Selection Performance

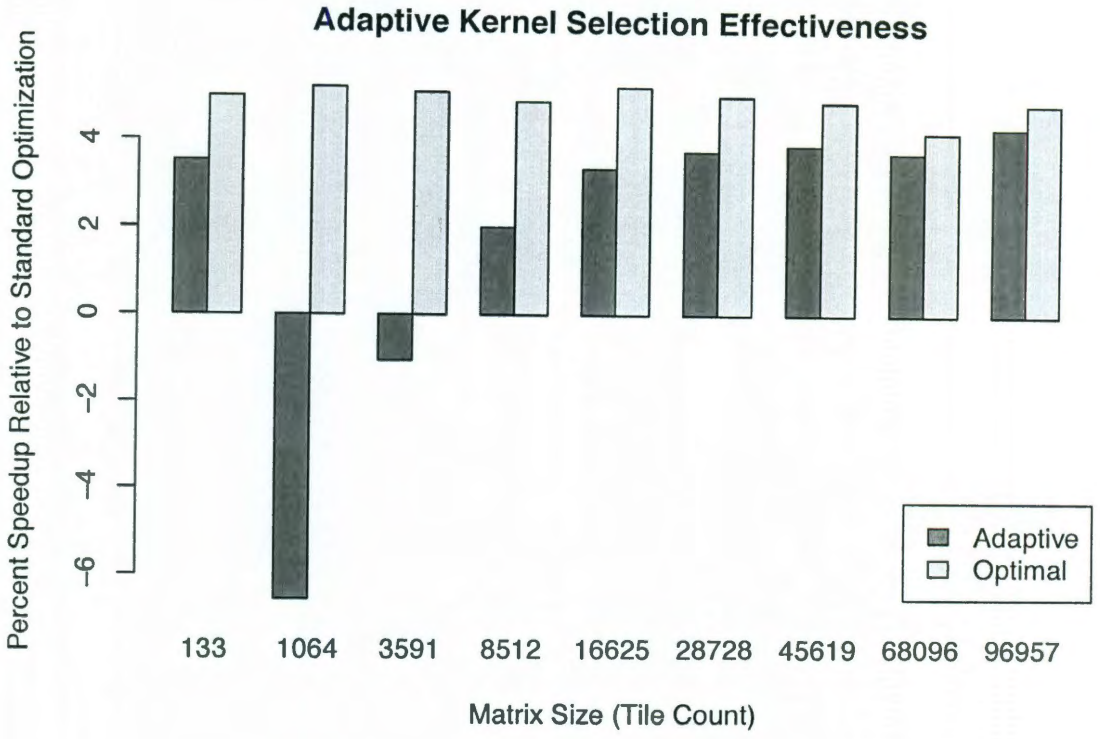


Figure 5.8: Adaptive Kernel Selection Effectiveness

algorithm speedup approaches the optimal speedup. The longer running times for the larger matrix sizes allow for the overhead to be amortized. As we move left, we see that the adaptive algorithm's speedup declines relative to the optimal speedup. For matrix sizes of 1064 and 3591 the adaptive algorithm actually exhibits a slowdown relative to the standard optimization sequence. Recall that the epoch size is 1000 iterations, however—these smaller matrix sizes only allow for approximately 1 and 3 epochs of adaptation, respectively. It isn't reasonable to expect the adaptive algorithm to be able to achieve a speedup in that amount of time. Perhaps decreasing the epoch size or adjusting the alpha factor that controls the rate at which kernel allocations change could allow the adaptive algorithm to achieve better results for these matrix sizes.

The last point of interest for Figure 5.8 is for the matrix size of 133 tiles. The adaptive algorithm for this matrix size actually exhibits a better speedup than for the

next two larger sizes. The explanation is quite simple: 133 tiles represents only about 13% of the first epoch, during which only the first few kernels will be exercised. Recall from Figure 5.7 that the first few kernels, for this example, happen to use the most effective unroll factors. Therefore, by chance, the adaptive algorithm unknowingly selects the best kernels for this matrix size.

Finally, let's consider the *efficiency* of the adaptive kernel-selection algorithm, as shown in Figure 5.9. We define efficiency as the percent of optimal speedup that the algorithm achieves. This chart is just a variation on the last chart, Figure 5.8, which shows both the achieved speedup and the optimal speedup. The efficiency is computed by dividing the achieved speedup by the optimal speedup—an efficiency of 100% indicates that the algorithm achieved the optimal speedup, while an efficiency of 0% indicates that the algorithm achieved no speedup compared to the standard optimization. A negative efficiency indicates that the algorithm caused a slowdown compared to the standard optimization. The efficiency for the left-most three matrix sizes follows directly from the discussion in the previous paragraph. Also, the efficiency metric increases as the matrix size increases; this makes sense, as the longer running-time amortizes the algorithm's overhead. However, the most interesting observation from this chart is that the efficiency seems to approach 89% and then level off. This number corresponds to the winner's bonus parameter used in the readjustment of allocation weights. The epoch winner has a target allocation that asymptotically approaches 90% of the iterations, while the remaining 10% are distributed among the other kernels. The epoch winner, even after many consecutive epochs as the winner, will always max-out at an allocation of 90%. Since most of the other kernels in this case study achieve a slowdown compared to the standard optimization, those 10% of iterations will bring down the average and we can never expect to achieve an optimal efficiency of 100%. Perhaps we could increase the winner's bonus over time to allow for increased efficiency for long running applications.

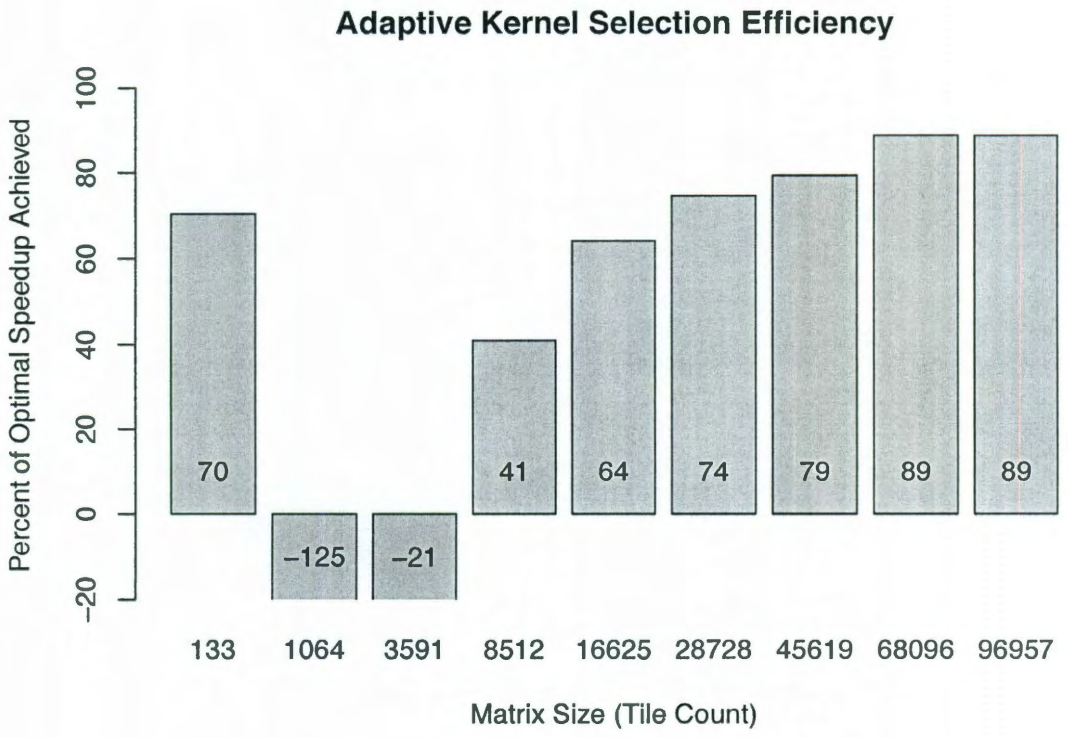


Figure 5.9: Adaptive Kernel Selection Efficiency

Or, we could allow the slowest kernels to slowly drop out of service. These retired kernels could either be permanently discarded, or they could be occasionally cycled back into service, one at a time, to allow for future adaptation while minimizing the performance overhead per epoch.

All results up to this point depict the performance of the adaptive code-selection algorithm when the kernels are statically generated and compiled. We can estimate the impact of dynamically generating the kernels by measuring the time required to statically compile the kernels—this should provide an upper bound, since dynamic compilation will be performed in-memory and require fewer disk accesses than static compilation. Figure 5.10 presents the estimated effectiveness of the adaptive code-selection algorithm when the cost of dynamic kernel generation is included. The profitability tradeoff shifts to the right, indicating that a longer running applica-

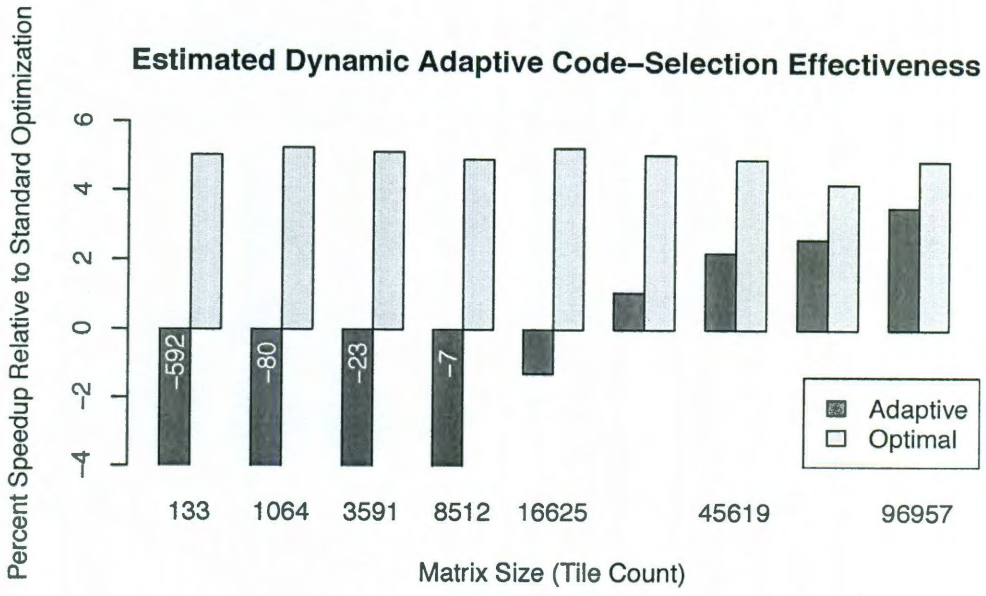


Figure 5.10: Estimated Dynamic Adaptive Code-Selection Effectiveness

tion is required to offset the increase in overhead. In fact, for small matrix sizes, the adaptive code-selection algorithm results in significant performance degradation. This degradation can likely be reduced by delaying the kernel generation until the application runs long enough to amortize the overhead. Also, the individual kernels can be generated one at a time rather than all at once, allowing the overhead to be distributed over time. For large matrix sizes, the overhead is offset and the adaptive code-selection algorithm is able to achieve a modest program speedup.

5.5 Related Work

The work presented in this chapter differs from much of the previous work on runtime optimization in two ways. First, prior work relies much more heavily on dynamic program transformation and code generation, while adaptive code-selection can rely solely on static code-generation; this difference allows adaptive code selection to avoid overheads associated with dynamic compilation. Second, prior work often often does

not evaluate the impact of dynamic optimization decisions, instead assuming that the optimizations will improve performance. Adaptive code-selection, on the other hand, directly evaluates the performance of various kernels and selects the fastest one.

Mars and Hundt present *Scenario Based Optimization* [36], an approach that is very similar to adaptive code-selection. Their approach statically generates a small number of application variants that are specialized for various runtime contexts that might be expected. Then, using hardware performance counters, their algorithm dynamically determines the scenario in which the application is running and selects the appropriate variant. Our approach is more general in the sense that it need not statically determine the specific scenarios for which to specialize—general parameter searching is possible.

Fursin *et al.* employ a similar technique for pruning search spaces for iterative optimization [26]. Traditional iterative compilation tests one optimization sequence per program invocation. Instead, Fursin *et al.* test multiple optimization sequences per program invocation by statically generating multiple variants and cycling between the variants at runtime. They are able to achieve similar performance benefits while significantly reducing the number of program invocations required. Our goal is similar to that of Fursin *et al.*, except that we constrain our algorithm to operate within a single program invocation. Thus, our adaptation must be more targeted and able to more quickly improve performance.

The idea of adaptively identifying and favoring the best kernel relates to the ideas used in adaptive clinical trials for medical research [10]. Adaptive clinical trials use accumulated results to guide the future direction of the trial. For example, when a drug or dose begins to exhibit negative outcomes, such as unexpected side-effects or poor patient-response, fewer new patients will be assigned to that group. On the other hand, groups with drugs or doses that exhibit positive outcomes will begin to receive a larger share of the new patients. The goal of these approaches is to

improve overall patient outcome while still maintaining statistical integrity for the clinical trial. Adaptive code-selection pursues a similar goal of improving the overall runtime of an application, but the constraint of maintaining statistical integrity is less critical. Adaptive clinical trials have shown that Bayesian statistical methods are an effective technique for summarizing trial behavior and guiding future outcomes [9]. Although such techniques might ultimately prove too computationally expensive for online use in an adaptive code-selection framework, it may still be worth exploring as an approach that achieves more fine-grained adaptation than the current algorithm achieves with epoch-based adaptation.

5.6 Conclusion

This chapter presented and evaluated an algorithm for dynamic, adaptive code-selection, a technique intended to reinforce cases in which a static compiler is unable to deliver sufficient performance. Adaptive code-selection allows a program to empirically tune its performance throughout execution by automatically identifying and favoring the best performing variant of a routine. This design directly supports the overall goal of this thesis—improving a compiler's adaptability—by allowing a compiler to empirically tune its optimization decisions for the observed behavior of a running application. The case study presented in Section 5.4 indicates that adaptive code-selection is an appropriate technology for dynamically choosing between different static-compilation strategies. Alternatively, this technique can be used with IR annotation, described in Chapter 4, for performing dynamic, feedback-directed optimization.

Chapter 6

Conclusions

Because computational science has become an essential tool for research in the natural sciences, it is imperative that scientists be able to utilize the full computational power available to them. Unfortunately, compilers often struggle to deliver the desired performance, forcing scientists to spend time manually tuning their application's performance rather than advancing their scientific research. Improving compiler technology should alleviate this development burden, but doing so requires addressing two critical challenges: (1) the underlying hardware platform is rapidly changing with advances in computer architecture and (2) the application software is difficult to statically model and predict. This thesis argues that improving the adaptability of compilers will help them overcome these challenges. The two main techniques presented in this dissertation—automatic resource characterization and selective, dynamic optimization—will allow the compiler to better adapt to both the rapid development cycle of computer hardware and the dynamic behavior and context of application software.

6.1 Automatic Resource Characterization

Automatic resource characterization, described in Chapters 2 and 3, comprises a suite of micro-benchmarks that empirically measure the performance-related characteristics of a particular system. These characteristics can be supplied to a parameterized compiler, allowing it to better target the specific features of that system. This thesis focused on characterizing a system's memory hierarchy, because this architectural feature has a significant impact on performance. We presented techniques for measuring a system's data cache and TLB capacity, line size, and associativity; techniques for measuring a system's instruction cache capacity and associativity; and techniques for detecting unified levels of cache.

The main concept behind all of the resource characterization micro-benchmarks is a simple inverse problem: treat the system as a black box and infer details about its composition based on the observed performance differences between a variety of a carefully constructed program kernels. The challenge is in the implementation details, however, and has led to several novel contributions. First, each characteristic should be discovered independently of the other characteristics, even if the characteristics are dependent in theory. Our approach creates distinct benchmarks for each characteristic, with minimal dependences between them, which minimizes the chance that an error in one benchmark will propagate through to other benchmarks. Second, the automatic interpretation of benchmark results is not straightforward, and cannot rely on arbitrary thresholds. Simple analysis techniques result in inconsistent results across a wide variety of test systems. Instead, we present a sophisticated analysis algorithm that borrows techniques from signal processing, statistics, and mathematical optimization. Following these strategies, we were able to correctly characterize the data cache and TLB parameters on over 20 test platforms that span a variety of architectures and operating systems.

In contrast, our work on characterizing a system's instruction cache has shown that

it is quite difficult, if not impossible, to correctly characterize the instruction cache using portable C code. We address several challenges in this context and present a benchmark that, although is able to identify all levels of the instruction cache on our test systems, suffers from occasional false-positive conclusions. We address scalability issues by modularizing the benchmark into a set of kernels, which are cyclicly linked through function pointers. The size of the instruction footprint can be controlled by modifying the head pointer in the kernel list. We present several techniques for estimating the code size of each kernel, since there is no portable method for measuring the code size of a C function. The main challenge with this benchmark is constructing a kernel body that consumes instruction memory sufficiently fast, since the processor's instruction-fetch mechanism will easily hide the fetch latency for a slow kernel body. We show that a carefully constructed machine-code version of the instruction cache benchmark can achieve more accurate results, but it relies on two properties that simply are not achievable in a C program: sparse and randomized execution patterns. Fortunately, we show that the unified cache benchmark, which relies on techniques from both the data cache and instruction cache benchmarks, still performs correctly despite the challenges encountered with the instruction cache benchmark.

The success of the resource-characterization micro-benchmarks presented in this dissertation should spur the development of parameterized implementations of classic machine-dependent compiler optimizations, such as loop tiling and instruction scheduling. Today, these transformations produce significant performance improvements, but require model-specific tuning to approach their full potential. If these transformations are appropriately parameterized, then the model-specific tuning can be replaced (or at least augmented) with the results of the resource-characterization benchmarks, greatly reducing the time and effort required to port a compiler to a new platform. In this manner, the resource characterization will enable the compiler

to automatically and quickly adapt to changes in hardware.

6.2 Selective, Dynamic Optimization

Although enabling a compiler to automatically adapt to new hardware is critical for improving compiler performance, this approach makes no headway in addressing the challenge of complex and unpredictable application behavior. This thesis argues that dynamic optimization is an indispensable reinforcing technology in an adaptable compiler, allowing the compiler to dynamically evaluate and adjust its optimization decisions at runtime. The application's observed behavior and runtime context allow the dynamic compiler to identify and selectively target regions of code for which the static compiler was unable to deliver sufficient performance. This thesis presented two techniques for enabling dynamic optimization: intermediate-representation (IR) annotation and adaptive code-selection.

6.2.1 IR Annotation

IR annotation, described in Chapter 4 is low-overhead approach for enabling selective, dynamic optimization. IR annotation trades code growth for performance and flexibility by maintaining multiple representations of the same program—a fully-optimized native binary is tagged with a higher-level compiler intermediate-representation of itself. This approach is designed to minimize the overheads associated with dynamic compilation by removing unnecessary work from the common path. The cost of the aggressive static optimization is incurred offline, but leveraged at runtime by the optimized machine code. The IR annotation is simply registered with a runtime system upon program invocation, but loading and parsing is delayed until the runtime system decides that optimization is profitable—only at that point is the cost of the dynamic compilation allowed to be incurred. Our results indicate that the base IR annotation

framework can be implemented with essentially zero performance overhead for the target applications. The main overhead incurred for this technique is code growth, which our results suggest is limited to a factor of approximately 2.5. Not only is this result understandable, since the program contains multiple representations of itself, it is quite reasonable given the decreasing cost and increasing capacity in modern storage technology.

Considering the reasonable costs of IR annotation, its advantages are noteworthy. It leverages an existing compiler infrastructure, eliminating the need to rewrite transformations for a dynamic compiler. It improves portability by pushing the platform-specific details into the compiler's code generator. The compiler IR is usually much higher-level than machine code, which enables more powerful optimizations than are possible with a binary optimization system. And, this strategy can support optimization across all languages for which the compiler provides a front end. When paired with a light-weight performance-profiling framework and a code generator that can be invoked at runtime, IR annotation has the potential to reduce the overhead and improve the overall outcome of dynamic compilation.

6.2.2 Adaptive Code-Selection

Adaptive code-selection, described in Chapter 5, is a dynamic technique that allows a running application to identify and select the best-performing function from a set of variants. The target of this technique, referred to as a kernel, can be any region of code that heavily repeats during a single program invocation. The kernel is replaced with a call to a control function that oversees the adaptive code-selection algorithm. Given a set of possible kernel variants, or kernel instances, the control function dynamically cycles each instance in and out of use, while collecting and maintaining performance feedback on each. Over time, the algorithm begins to identify the best performing kernels, which are then allocated larger fractions of the total running time. Since

the algorithm continues to monitor performance, it is able to adaptively adjust the kernel allocations to accommodate changes in program behavior or additional kernel instances that may be provided by a dynamic compiler.

In our case study with a matrix-multiply benchmark, we statically generated 32 kernel variants to which we applied loop unrolling with different unroll factors; each kernel was then compiled with standard compiler optimizations. When used in this manner, the adaptive code-selection algorithm essentially performs a search for the best unroll factor; any parameterized optimization can be tuned in this manner. Our results indicate that the adaptive code-selection algorithm was quickly able to identify the best kernel and realize an overall speedup for all but the shortest running invocations of the benchmark. Although the speedup from this case study was only about 4%, the adaptive code-selection algorithm was able to achieve up to 90% of the optimal speedup. The optimal speedup is defined as the total benefit possible if the algorithm had known ahead of time which kernel was the best. Since the achievable speedup is a function of the optimization performed, we can expect better speedups with more aggressive optimizations.

The most difficult challenge with adaptive code-selection is identifying a region of code that is suitable for targeting as a kernel. This thesis described several approaches for identifying a target kernel, but did not implement an automatic transformation for enabling adaptive code-selection. The case study was performed manually. Perhaps, as a first step, adaptive code-selection can be enabled manually by the programmer, through source-code annotations. The main obstacle for developing a fully automatic approach is that the amount of work performed per kernel invocation must be roughly fixed—otherwise, the adaptive code-selection algorithm will make unfair comparisons between different kernel invocations. If different kernel invocations perform different amounts of work, it may be possible to quantify the amount of work to allow normalization of the running times; it is unclear whether this can be done automatically.

Adaptive code-selection can be used with IR annotation and a dynamic compiler to implement feedback-directed optimization at runtime, or it can simply be used as a dynamic, adaptive technique to select between several different static compilation strategies. Regardless of whether the kernel variants are generated statically or dynamically, this technique improves a compiler's ability to adapt its optimization decisions to the observed behavior of a running application.

6.3 Future Work

There are many interesting ways that the research described in this dissertation can be continued in future work. This section briefly describes some of them.

6.3.1 Resource Characterization

There are several interesting future experiments that could improve our understanding of architecture design and its impact on a system's performance characteristics. It would be interesting to investigate the reasons why the effective cache sizes are often so much smaller than the actual cache sizes. Our results indicate that the virtual-to-physical mapping and the small page-size relative to the cache capacity are the main contributing factors. We could run experiments that test the impact of large pages on effective capacity; if the effective capacity grows with large pages, that would be a strong argument for using large pages.

The configuration space for data footprints, depicted in Figure 2.3 in Chapter 2, is actually a 2-dimensional search space. Our tests perform 1-dimensional sweeps in different directions though the 2-dimensional search space. For academic understanding, it would be interesting to perform a full 2-dimensional sweep across the entire search space. Ideally, a full sweep would present a much clearer picture of where the effective cache and TLB boundaries occur on a particular system. Also, the full sweep

might improve our understanding of access latency, since the effective access latency for a particular address depends both on the state the cache and the TLB, which can be different depending on the context of the access.

The characterization benchmarks presented in Chapter 2 probe a system's cache from the perspective of integer loads. It may be important to extend the benchmark to also characterize caches from the perspective of floating-point loads. For example, on the Intel Itanium processor the L1 cache does not contain floating-point values; instead, all floating-point accesses interface directly with the L2 cache [4]. Although our benchmark correctly characterizes this system's L1 cache, it does not recognize that floating-point values bypass the L1 cache. Similarly, it may be helpful to further distinguish between loads and stores, since the store latency may differ depending on whether the cache employs a write-back or write-through policy.

Finally, to isolate the impact of various hardware features on the memory hierarchy behavior, we could apply our microbenchmark to a cache simulator. This would allow us to precisely know and control all of the underlying variables of the system. For example, we could directly compare the differences between two otherwise identical systems, except that one performs hardware prefetching and the other does not.

6.3.2 Dynamic Optimization

Since IR annotation is a foundational technology for enabling dynamic compilation and optimization, much of the future work will focus on identifying uses for this technology and applying it. Chapter 5 provides one possible application of IR annotation, but other applications undoubtedly exist. Unfortunately, the engineering costs for building a full runtime-optimization system may be quite high, since doing so requires providing a light-weight performance profiler and a compiler that supports runtime code-generation. Leveraging existing technology, such as HPCToolkit [2] and LLVM [33], may help reduce the implementation costs.

Further work can be done to reduce the code growth caused by IR annotation. First, it may not be necessary to preserve IR for every function in an application. For example, debugging, logging, or error-handling routines may not exhibit significant benefit from runtime optimization. Instead, the compute-intensive loops and routines should be the target of IR annotation. An easy solution is to allow developers to specify, through source-code annotations, the routines for which IR should or should not be preserved. A more sophisticated approach might try to statically determine which routines are likely to benefit from runtime optimization, only preserving IR for those routines.

A second technique for reducing code growth is to employ various lossless compression techniques on the annotated IR. Preliminary experiments suggest that using `gzip` compression [20, 21] on the SPEC benchmarks can result in an average reduction in code growth of about 25%. Incorporating such compression algorithms into the static IR annotation pass is straightforward, and the cost of decompression would not be incurred until the runtime system decides to load the IR for optimization. That is, the performance overhead of the base runtime system should not be increased by compression.

Third, the IR modules may contain redundant information that can be pruned to reduce code growth. Each IR module creates its own symbol table of all symbols that are referenced in a module. If multiple modules reference the same symbol, then there will be redundant entries in the symbol tables. By applying LLVM's link-time-optimization, we will be able to generate a single symbol table and embedded IR module for each application and shared library. This should eliminate much of the symbol table redundancy. Additionally, program constants and their initialization values are preserved in the embedded IR, even though the initialization value is also available in the program binary. These constants can safely be removed from the IR, since the value can always be queried from the executable's constant-data section.

There are several ways in which the adaptive code-selection research can be continued. First, several parameters—such as starting or minimum epoch size, maximum epoch size, winner's bonus, and alpha factors that control the rate of change in the moving averages and iteration allocations—control the behavior of the adaptive kernel-selection algorithm. Since many of these parameters were arbitrarily chosen for the case study in Section 5.4, it would be interesting to perform further studies on the impact of these parameters. These parameters aren't necessarily intended to be modified by the application developer (although they could be, if the developer is knowledgeable about the adaptive kernel-selection algorithm). Instead, a better understanding of the parameters may allow us to improve the algorithm. It seems unlikely that one particular set of parameter values will be optimal for all scenarios; rather, the algorithm may benefit from using different parameters for different applications and systems. For example, an application with very consistent kernel behavior may benefit from a larger alpha factor, allowing the iteration allocations to respond more quickly to changes in kernel performance. On the other hand, applications with more variation between the performance of each kernel invocation may require smaller alpha factors to allow the algorithm to thoroughly evaluate a kernel before responding. Perhaps the algorithm can be modified to allow the parameters to adaptively change, within limits, similar to the way that the epoch size changes.

Much work remains to fully automate the process of identifying and extracting kernels. The profitability calculation may be problematic—it may be quite difficult for the static compiler to determine whether a region of code is an appropriate kernel candidate. It may be possible to aggressively identify kernels statically, but determine at runtime whether the adaptive kernel-selection algorithm should be applied. A compromise to full automation might be to introduce compiler directives that allow the developer to specify the kernel boundaries with source-code annotations.

More research needs to be done to identify other compiler optimizations that

can be tuned with adaptive kernel selection. Optimizations that use adjustable parameters are obvious candidates—graph coloring register allocation may be a good example. Also, optimizations that select from among a set of different strategies, such as list scheduling, may be good candidates. Loop tiling is definitely an optimization that may be parameterized, to select effective tile sizes, but this optimization differs from others because it can generate a single version of parameterized code rather than many different versions of code. Tile-size selection can still leverage the adaptive code-selection framework for selecting between different tiles sizes, but this use would not need more than one kernel variant.

Chapter 5 focused on a case study where work per kernel invocation remained constant throughout the entire execution. Section 5.2 identified other techniques for handling kernels that perform different amounts of work per invocation: (1) divide the kernel into smaller, fixed-size units of work; (2) quantify the amount of work per invocation and compute a normalized metric for performance; (3) rely on statistical averaging by collecting many sample kernel-invocations before drawing any conclusions. All of these techniques are interesting areas of future research that require further study to explore their usefulness.

Although this thesis focused on applying adaptive code-selection in a stable and unchanging environment, it would also be very useful for allowing an application to adapt to external factors that influence performance. As the number of cores on multi-core architectures grows, so does the likelihood that an application will have to share various system resourced with other applications. The best kernel for a particular algorithm may vary significantly, depending on a system's load and each individual application's behavior. The adaptive-code selection algorithm would allow an application to dynamically adapt its kernel to best match the available system resources. Similarly, architectures that allow various resources to be dynamically modified or disabled may be good candidates for adaptive code-selection. For example, an oper-

ating system may dynamically scale a processor's clock frequency or modify it's cache configuration to save power. It is unlikely that the same kernel would perform the best for these vastly different contexts; instead, the adaptive code-selection algorithm would allow an application to adapt to these system-context changes by automatically migrating to the kernel that is best suited for the current context.

Finally, the case study in Chapter 5 only focused on applying the adaptive kernel-selection algorithm on statically-generated kernels. Although the results clearly indicate that adaptive kernel-selection is an appropriate technique for dynamically selecting between various static-compilation strategies, further work remains for performing dynamic feedback-directed optimization. This area of work would apply to cases in which the kernel variants are unknown at compile time or the number of variants is too large to compile statically. In these cases the kernel variants would need to be generated at runtime, with a dynamic compiler.

Bibliography

[1] Cca-forum. http://www.cca-forum.org.

[2] Hpctoolkit. http://hpctoolkit.org.

[3] Scientific discovery through advanced computing. Technical report, Office of Science, U.S. Department of Energy, March 2000.

[4] *Intel Itanium 2 Processor Reference Manual*, May 2004. Order Number: 251110-003.

[5] Ali-Reza Adl-Tabatabai, MichałCierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *SIGPLAN Not.*, 33(5):280–290, 1998.

[6] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[7] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, HPDC '99, pages 13–, Washington, DC, USA, 1999. IEEE Computer Society.

[8] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[9] Donald A. Berry. Bayesian clinical trials. *Nature Reviews Drug Discovery*, 5:27–36, January 2006.

[10] Donald A. Berry. Adaptive clinical trials: The promise and the caution. *Journal of Clinical Oncology*, 29(6):606–609, 2011.

[11] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2000.

[12] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 129–141, New York, NY, USA, 1999. ACM.

[13] C. Chambers and D. Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.

[14] Andrew Cleary, Scott Kohn, Steven G. Smith, and Brent Smolinski. Language interoperability mechanisms for high-performance scientific applications. In *SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, Yorktown Heights, NY, 1998.

[15] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. *The Journal of Supercomputing*, 36:135–151, May 2006.

[16] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23:7–22, August 2002.

[17] Tamara Dahlgren, Thomas Epperly, Gary Kumfert, and James Leek. *Babel Users' Guide.* Lawrence Livermore National Laboratory, November 2007.

[18] Anshuman Dasgupta. *Tailoring traditional optimizations for runtime compilation.* PhD thesis, Rice University, 2006.

[19] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, New York, NY, USA, 1984. ACM.

[20] P. Deutsch. RFC 1951: DEFLATE compressed data format specification version 1.3. Network Working Group, May 1996.

[21] P. Deutsch. RFC 1952: GZIP file format specification version 4.3. Network Working Group, May 1996.

[22] Jack Dongarra, Shirley Moore, Philip Mucci, Keith Seymour, and Haihang You. Accurate cache and tlb characterization using hardware counters. In *Proceedings*

of the International Conference on Computational Science (ICCS), pages 432–439, 2004.

[23] Alexandre X. Duchateau, Albert Sidelnik, María Jesús Garzarán, and David Padua. P-ray: A software suite for multi-core architecture characterization. *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Edmonton, Canada, July 31 - August 2, 2008, Revised Selected Papers*, pages 187–201, 2008.

[24] Basilio B. Fraguela, Yevgen Voronenko, and Markus Puschel. Automatic tuning of discrete fourier transforms driven by analytical modeling. *Parallel Architectures and Compilation Techniques, International Conference on*, pages 271–280, 2009.

[25] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[26] Grigori Fursin, Albert Cohen, Michael O'Boyle, and Olivier Temam. Quick and practical run-time evaluation of multiple program optimizations. In *Transactions on High-Performance Embedded Architectures and Compilers I*, volume 4050 of *Lecture Notes in Computer Science*, pages 34–53. Springer Berlin / Heidelberg, 2007.

[27] Jorge González-Domínguez, Guillermo L. Taboada, Basilio B. Fraguela, María J. Martín, and Juan Touriño. Servet: A benchmark suite for autotuning on multicore clusters. In *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, USA, April 2010.

[28] Alexander Grosul. *Adaptive ordering of code transformations in an optimizing compiler.* PhD thesis, Rice University, 2005.

[29] Mary W. Hall, Ken Kennedy, and Kathryn S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 424–434, New York, NY, USA, 1991. ACM.

[30] Ralf Holly. A reusable duff device. *Dr. Dobb's Journal*, pages 73–74, August 2005.

[31] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2010. Order Number: 248966-022.

[32] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing.* Portsmouth, VA, 2001.

[33] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[34] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:1–24, April 2004.

[35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[36] Jason Mars and Robert Hundt. Scenario based optimization: A framework for statically enabling online optimizations. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 169–179, Washington, DC, USA, 2009. IEEE Computer Society.

[37] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.

[38] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[39] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspotTM server compiler. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[40] Juan-Carlos Perez and Enrique Vidal. Optimum polygonal approximation of digitized curves. *Pattern Recogn. Lett.*, 15(8):743–750, 1994.

[41] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990.

[42] Tim Robertson, F.T. Wright, and R.L. Dykstra. *Order Restricted Statistical Inference.* John Wiley @ Sons Ltd., 1988.

[43] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and tlb performance and their effect of benchmark run times. Technical Report UCB/CSD-93-767, EECS Department, University of California, Berkeley, Aug 1993.

[44] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and tlb performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, 1995.

[45] Jeffrey A. Sandoval. Tuning an adaptive-compilation search space with loop unrolling. Master's thesis, Rice University, 2007.

[46] R. G. Scarborough and H. G. Kolsky. Improved optimization of fortran object programs. *IBM Journal of Research and Development*, 24(6):660, November 1980.

[47] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, 2001.

[48] Nathan Russell Tallent. Binary analysis for attribution and interpretation of performance measurements on fully-optimized code. Master's thesis, Rice University, 2007.

[49] Reid Edmund Tatge, Alan Lee Davis, and Alan Scott Ward. Program optimization with intermediate code. U.S. Patent 20050125783, June 2005.

[50] Todd Waterman. *Adaptive compilation and inlining.* PhD thesis, Rice University, 2006.

[51] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. *SIGMETRICS Perform. Eval. Rev.*, 33(1):181–192, 2005.

[52] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, page 168, Washington, DC, USA, 2005.